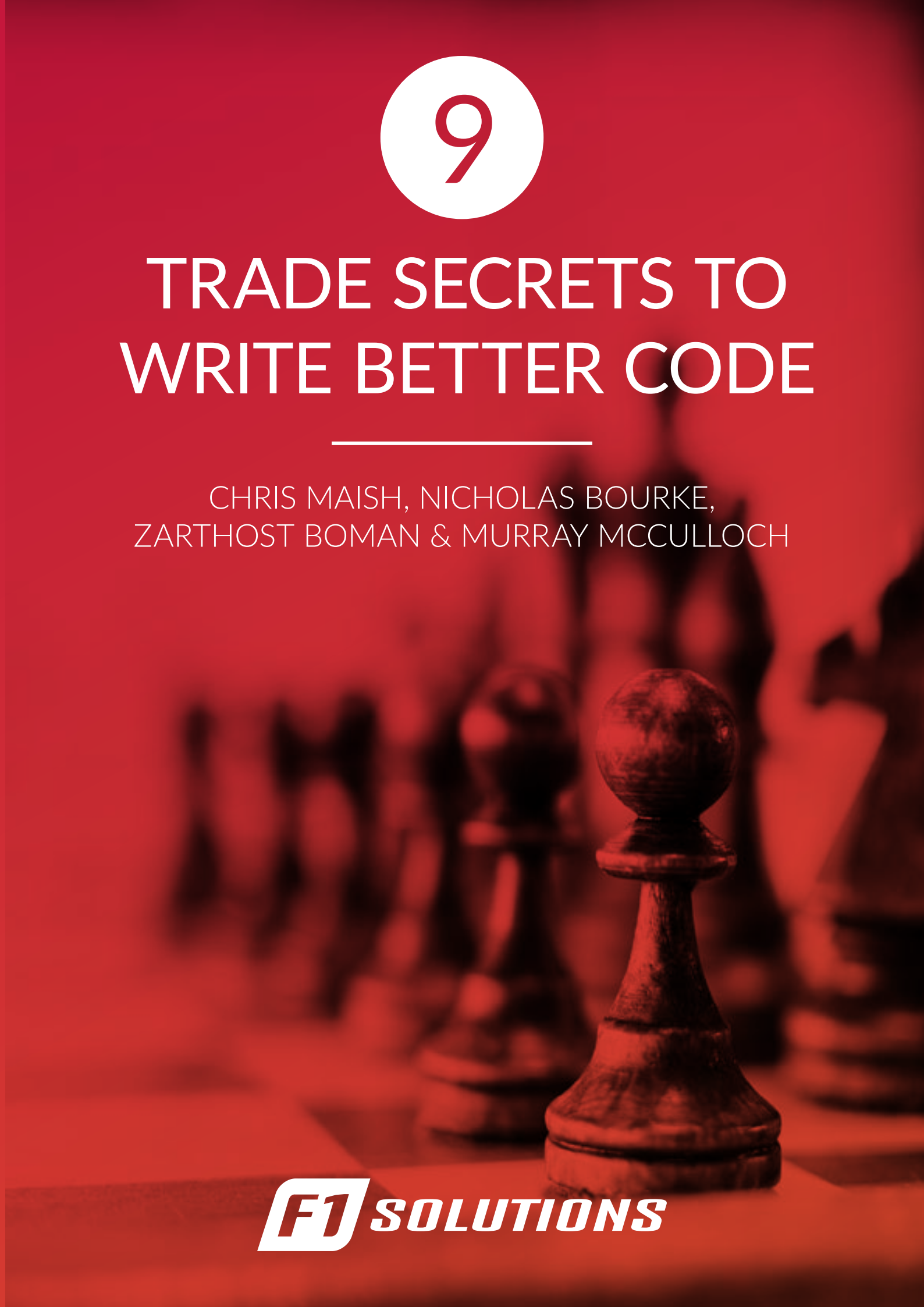
9

TRADE SECRETS TO WRITE BETTER CODE

CHRIS MAISH, NICHOLAS BOURKE,
ZARTHOST BOMAN & MURRAY MCCULLOCH



F1 SOLUTIONS

ABOUT THIS BOOK

Much has been written about Software Quality, mostly in an academic context.

For the purposes of this document, software quality refers to a number of separate concepts:

// Fitness for purpose (how well the software works, number of defects).

// Robustness, performance, security.

// Ease of comprehension and understanding from a developer/analysis perspective.

// Ease of comprehension and understanding from a user perspective.

To restate, high-quality software is software that:

// Consistently performs the tasks that it is designed to perform

// Has minimal defects

// Is secure

// Operates efficiently

// Is maintainable

One of the key steps to improve the quality of the software that you deliver is to build on a solid foundation, improving the quality of the underlying code.

This book focuses on improving the quality of the codebase itself, however code quality is just one piece of the puzzle — it must go hand-in-hand with management processes that work, alongside real technical skills. These three disciplines come together to allow teams to successfully execute on technical plans quickly and correctly.

ABOUT F1 SOLUTIONS

For over 20 years F1 Solutions has been building quality software solutions for Federal and State Government departments, small and large not-for-profits, and businesses in Canberra and across Australia.

Clients include the National Health Medical and Research Council (NHMRC), Cancer Institute of NSW, Department of Defence, Department of Social Services, St Vincent de Paul Society, Elections ACT, ARENA, Sugar Research Australia, Australian Pork Limited — just to name a few!

ABOUT THE AUTHORS

CHRIS MAISH

Chris Maish is a Solution Architect at F1 Solutions. He's worked with us for over 7 years and has worked on projects, including OmniStar, DAFF ARC and iTravel. Chris holds a MCTS in Web Application Development with the Microsoft .NET Framework 4 and has over 10 years' experience in software development. If Chris didn't go into software, he says he'd probably have studied to become a composer, an aerospace engineer or a professional juggler.

NICHOLAS BOURKE

Nicholas Bourke is the Development Team Lead at F1 Solutions. He's worked here for over 6 years and has worked on some of our most exciting projects, including OmniStar, eLAPPS, and NAATI SAM. Nick holds a Bachelor of Information Technology from the Australian National University, and has over 6 years' experience in software development. If Nick wasn't a software developer he says he would have become an Archaeologist.

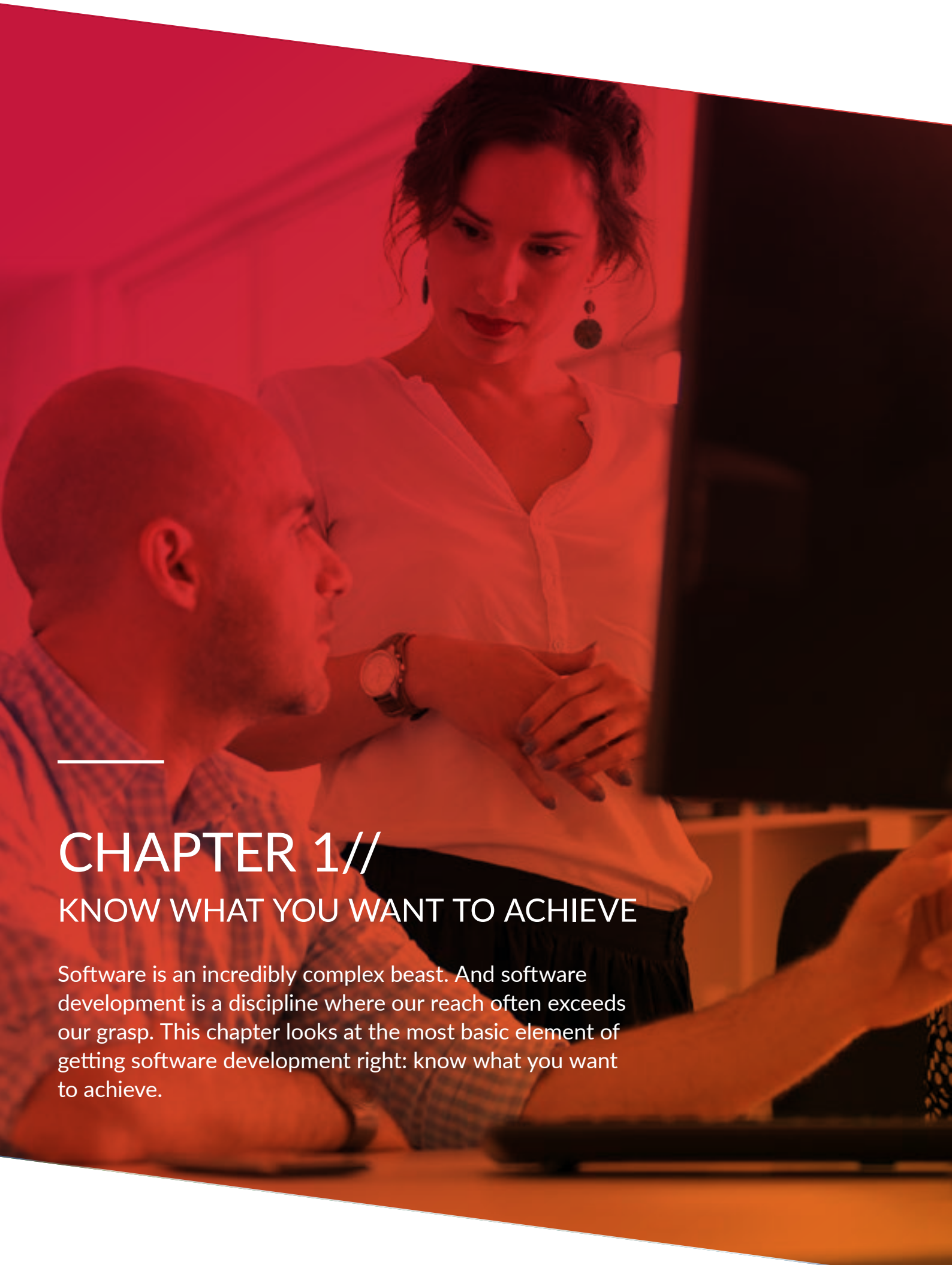
ZARTHOST BOMAN

Zarthost Boman is the Director of Operations at F1 Solutions. He has over 16 years' experience in software development and has worked here for over 9 years. During that time he has overseen all of our major projects. Zar holds a Bachelor of Information Technology and Graduate Diploma of Business Administration. His favourite part of software development is designing and implementing user interfaces, especially in web applications. He says there is nothing better than coming up with a slick design, coding it and having it look great whilst still being functional and intuitive to use.

MURRAY MCCULLOCH

Murray McCulloch is a Senior Software Developer at F1 Solutions. He's worked here for over 3 years and he has had a big involvement in the development of our OmniStar solution. He has over 20 years' experience in software development and has worked in nine jobs, eight cities, four states and two countries. He uses that experience to mentor and manage our more junior staff.

CONTENTS



CHAPTER 1//
KNOW WHAT YOU WANT TO ACHIEVE

Software is an incredibly complex beast. And software development is a discipline where our reach often exceeds our grasp. This chapter looks at the most basic element of getting software development right: know what you want to achieve.

SOFTWARE IS AN INCREDIBLY COMPLEX BEAST. AND SOFTWARE DEVELOPMENT IS A DISCIPLINE WHERE OUR REACH OFTEN EXCEEDS OUR GRASP. THE NATURE OF OUR INDUSTRY IS SUCH THAT SHORT DEADLINES, LOW BUDGETS AND LARGE (AND MOVING) SCOPES ARE COMMONPLACE.

In the absence of infinite timeframes and budgets, defects are, simply put, a reality. The trick with developing quality software is to try to avoid the common and unnecessary problems. This allows QAs more time to focus on non-obvious problems, which in turn maximises quality and helps to minimise the overall cost of building and maintaining the system. How you go about this is highly dependent on its purpose, importance and intended usage.

TIP: Developing fully defect-free software is an expensive and near-impossible task, which simply isn't possible in most situations. You will need to make compromises when it comes to elegance, cost, performance and quality.

To find the ideal sweet spot, we find that it's always important to manage scope closely. Managing the other variables is something that comes with experience, along with a good understanding of the business and system. You need to know what you want to achieve but this isn't always as simple as it sounds.

When we build software systems, everyone immediately asks the question: "What would we like the software to do?" This is an important question, but it's certainly not the only question that should be asked.

Another question often forgotten is: "How would we like to maintain the software?" A solution which does what the business wants but is completely unmaintainable is not considered high quality. That's why we favour simpler, clearer (and therefore more maintainable) code over more performant code. (Unless, of course, there is a demonstrated need for performance.)

On the other hand, there are situations where code does not need to conform to strict quality requirements; for example code that is intended to be used only a handful of times and by skilled staff. While it is certainly better for this code to be high quality, the trade-off between functionality/cost and quality may lay closer to being developed quickly.

Code that falls into this category includes:

// One-off migration scripts,which are usually judged based on their success

// Developer environment utilities,which often have a moderate level of churn and serve no purpose outside of the development team

// Prototypes

REQUIREMENT ELICITATION

While there naturally needs to be a lot of room for variation in requirements (and the resulting system), it is advantageous to ensure that you have the most accurate and complete possible picture of the desired solution. This has two meanings:

1. You need to know what you're building in the first place.

2. You don't want to fundamentally change what you're building part-way through.

If you don't know what you're building, a lot of rework will be necessary. Rework is where codebases become messy. Knowing what you're writing, when you first write it, will minimise rework and time directly involved. It will also keep the codebase neat, tidy and well structured.

On a broader level this is also true. A system designed to process invoices is not a good system for creating and managing pictures and videos. Having a good grasp on requirements means later on you won't be trying to fit a round peg into a square hole.

Once you have a good grasp on the functional requirements, get a good grasp on the non-functional requirements. These are equally difficult to incorporate into a system. It's important to plan for the needs of the system as it moves into the near-to-mid future — don't just focus on the needs of the business today.

A design for a system which is intended to support 100 users is very different to that intended to support 10,000. While smaller systems can easily be vertically scaled there is a point where a different approach is necessary.

As things scale up, the programming designs change pretty significantly — synchronous, centralised and procedural approaches are replaced by asynchronous, service-oriented and event-driven approaches — at these scales, messages, job queues and workers will become your bread and butter.

Similarly, the types of issues encountered typically become more complex (i.e. race conditions, localisation errors and others take the place of more basic logic errors).

SOLUTION AND PRODUCT DESIGN

Solution design is a complex and in-depth task that requires a lot of skill and expertise.

Thousands of books have been written on this topic alone.

What we instead want to focus on is how to design a solution in a way that minimises basic errors.

From a code quality perspective, a good solution design will:

// Use suitable technologies for the problem at hand

// Use standard technologies for the problem at hand

// Avoid common security pitfalls

// Avoid common code issues

// Be structured in a way that makes sense to developers

// Try to hide subsystem complexity from external consumers

// Take into account non-functional requirements (including future growth)

// Be simple enough to explain in a few minutes

// Be "boring"

// Be extensible

As a general rule, when picking technologies, favour compilable, transpilable or lintable code.

Pick a stack that, by design, avoids placing data into global scope.

Try to pick technologies that are easily refactorable (refer to Chapter 10 for more details on refactoring) and that allow for remoting with a minimum of fuss.

One of the fundamental mistakes Microsoft made with the ASP.NET webforms stack was that they tried to abstract away the internet layer.

This led to simpler code for developers with experience developing non-web applications, such as winforms, but "went against the grain" of web development.

But this abstraction often led to slow and bloated pages which were wasteful with bandwidth.

We have seen cases where applications were transferring hundreds of kilobytes of data between the client and server on each request needlessly.

It's possible to turn many of these things off, but the design itself is fundamentally broken by default.

As such, unless you need what the abstraction provides or you gain some significant advantage from it, you should avoid the technology completely.

We would consider these types of pitfalls to be similar to those around certain types of falsey values in languages like JavaScript, or random library method naming and argument ordering in PHP.

Both JS and PHP are languages which grew organically and have problem areas — JavaScript has a seminal book called, *JavaScript: The Good Parts*, which guides readers away from the language and library pitfalls; PHP has had much written about its shortfalls, for instance this blog.

That's not to say that these languages are on their own poor choices for achieving goals — in the case of JavaScript, it's the only way to achieve some goals —it's that these languages carry significant risks for most teams that must be weighed against their benefits.

An area where many of the .NET components shine is in being secure by default. It is a more difficult task for a beginner developer to inadvertently compromise the security of an ASP.NET website than with some other technologies (e.g. PHP).

On the next page is an example of the two pieces of code.

C#/ SQLCLIENT

```
string commandText = "SELECT firstname, lastname, address, age FROM friends WHERE
firstname = @firstname And lastname = @lastname";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    SqlCommand command = new SqlCommand(commandText, connection);
    command.Parameters.AddWithValue("@firstname", firstname);
    command.Parameters.AddWithValue("@lastname", lastname);

    try
    {
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (reader.HasRows)
            {
                while (reader.Read())
                {
                    Console.WriteLine("{0}\t{1}\t{2}\t{3}",
                        reader.GetString(0),
                        reader.GetString(1),
                        reader.GetString(2),
                        reader.GetInt32(3));
                }
            }
            else
            {
                Console.WriteLine("No rows found.");
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

PHP/ MYSQL_QUERY

```
// This is the best way to perform an SQL query
// For more examples, see mysql_real_escape_string()
$query = sprintf("SELECT firstname, lastname, address, age FROM friends
    WHERE firstname='%s' AND lastname='%s'",
    mysql_real_escape_string($firstname),
    mysql_real_escape_string($lastname));

// Perform Query
$result = mysql_query($query);

// Check result
if (!$result) {
    $message  = 'Invalid query: ' . mysql_error() . "\n";
    $message .= 'Whole query: ' . $query;
    die($message);
}

// Attempting to print $result won't allow access to information in the resource
// One of the mysql result functions must be used
// See also mysql_result(), mysql_fetch_array(), mysql_fetch_row(), etc.
while ($row = mysql_fetch_assoc($result)) {
    echo $row['firstname'];
    echo $row['lastname'];
    echo $row['address'];
    echo $row['age'];
}

mysql_free_result($result);
```

WHY C# IS BETTER SUITED

We believe that the C# code is better suited for this purpose for many reasons:

// The use of the general string interpolation function in PHP is concerning from a security perspective. There will be no complaints if you forget to call the mysql_real_escape_string method. Of course, you could manually interpolate the string in .NET, however this is a bigger error than forgetting to call an escape method.

// The static typing in C# makes its usage of the result of the query more obvious.

// The use of global methods in PHP result in the correct method to retrieve results not being fluently discoverable.

// The return value from the C# DataReader.Read is a boolean, which is far clearer in intent than an object.

// The garbage collection (GC) and scoping via using means that you don't need to worry about freeing the DataReader in C#.

// C# being compiled means that you'll pick up errors in code syntax (and in actually using the data that was extracted).

It's less likely that you're going to make mistakes — security or otherwise — and it is easier to write code that works well.

We are not the only ones who think this way, software developer, Dan McKinley, makes a good argument for what boring technology is and why you should use it in his blog post, *Choose Boring Technology*.

Anecdotally, the concept of "innovation tokens" is one that we have found to be the case. It is difficult to be successful with projects when you spend a lot of time putting out fires in the underlying platforms.

PROTOTYPING

Prototypes are one exception to the "minimise rework" rule.

As developers, we often create simplistic proof of concepts to show something is indeed possible.

These demonstrations are usually built with the minimum possible scope, incredibly quickly and cheaply, and with a minimal acceptable quality.

This is a good thing because they are not intended to be maintained, and should cost the business as little as possible (in terms of money and time) to produce.

The underlying code has usually taken multiple different approaches and carries some significant technical debt.

As such, it is almost always an error to turn around and put this code into a stable codebase which you intend to maintain simply because shifting quality is an incredibly difficult task.

TIP: When incorporating a proof of concept into existing code, start from scratch. Take the lessons learned from the proof of concept and architect a new solution based on what works, resisting the urge to copy-paste.

Prototypes are useful and important to the development process. They serve two purposes:

1. Show that a particular approach to solving a problem works
2. Allow people to see a system in operation.

Both of these help you get a better grasp of what you need to build.

Note: there is a difference between a prototype proof of concept and a minimal viable product, which brings us to our next point.

MINIMAL VIABLE PRODUCT

A Minimal Viable Product (MVP) is different to a prototype in that it's not a testbed for new ideas — it's a production-quality codebase.

When you are developing a MVP, you should consider quality as you would in a real system.

It is often desirable to create a prototype, and then to distil it into a Minimal Viable Product.

When you're starting on a new project, you should decide whether you're building a prototype or a product, then take the appropriate approach.

TECHNICAL DEBT

Technical debt is a pretty common term referring to the informal ledger of quality within a project.

Any time that a compromise is made on quality, technical debt is accrued. Each change that is made either increases or decreases the debt within the project.

Debt is also a good analogy because, over time, this debt takes on interest. The sooner you get to solving design problems and paying off debt, the easier it is.

One of the key ways to minimise technical debt involves trying to make changes that improve the structure, readability and simplicity of the underlying code; and trying to avoid making changes that reduce the readability and simplicity of the underlying code.

Another way to minimise technical debt is to know what you're making as early as possible in the process so that you reduce the number of decisions and changes that need to be made after the initial design.

You can also consider adding comments in code to flag areas that need to be revisited and refactored and add a task to the backlog to improve these areas. This way these areas are kept on the radar and considered for inclusions in upcoming sprints and iterations.

TIP: Pay off technical debt as soon as possible. If you need to get a change out quickly, get an initial fix out but perform the actual fix right away, otherwise you will be doing the same work twice.

ADDRESSING THE RIGHT CHALLENGES

Greenfield and Brownfield development are two different beasts.

Greenfield development involves a process of eliciting requirements from the business, designing an overarching system design, prototyping various pieces of the puzzle and then integrating them into the system design.

Brownfield development, on the other hand, largely focuses around refactoring and the ongoing management of technical debt while fixing existing problems within the design and codebase.

The significant differences between the two pose individual challenges when trying to maintain code quality.

This is an important realisation when stepping into development as it fundamentally changes many of the things that we want to achieve in a high-quality codebase.

Note: when adding new functionality to an existing system, this functionality will likely suffer from similar challenges to those that occur when developing a new system.

The table on the next page lists some general issues you are likely to stem from the different sources.

As can be seen from the table, with new systems most defects are introduced by internal factors. On aging systems, defects typically are introduced by external factors.

TIP: Because the types of defects encountered in new developments are not the same as those encountered in existing systems, different actions should be taken to ensure quality in the two different types of codebase.

Broadly speaking, internal factors are things that can generally be mitigated with improved developer tooling, training and expertise.

External factors are usually much more difficult to automatically mitigate, so instead your focus should be on monitoring and identifying problems and addressing them quickly when they occur.

It's a proactive/reactive split.

Additionally, the types of issues that are found in new systems are more obvious and replicable.

Those found in long-standing systems tend to be rare or non-replicable; things that can be worked around; or larger design changes.

This is manifested in old systems "showing their age", where issues tend to be more complex and expensive to fix.

This, along with the natural and ongoing accrual of unpaid technical debt is what leads to most product rewrites.

GREENFIELD DEVELOPMENT	BROWNFIELD DEVELOPMENT

SPECIFICATION DEFECTS
// Ambiguous requirement errors
// Missed edge-cases
// Unanticipated side-effects
	LEGACY DEFECTS
	// Breaking changes in underlying components
	// Technology deprecation

ARCHITECTURE AND DESIGN DEFECTS
// Unsuitable system design choices
// Non-functional requirements not being met
// Last-minute requirement changes
	SECURITY ISSUES
	// Fixes for newfound exploits
	// Replacement of outdated algorithms

SYSTEM DEFECTS
// Missed requirements
// Logic errors
// Timing defects/race conditions
// Security issues
	SCALING PROBLEMS
	// Existing design now unsuitable
	// Performance bottlenecks

UI DEFECTS
// Typos
// Client incompatibilities
	ISSUES INTRODUCED WITH NEW FUNCTIONALITY
	// Greenfield development issues

BASIC CODE DEFECTS
// Issues with code structure
// Syntactical issues in interpreted code
	REFACTORING ISSUES
	// Unintended functional changes
	// Introduced side effects

PROCESS PROBLEMS
// Defects introduced by inexperienced staff
// Syntactical issues in compiled code	

PICK YOUR BATTLES FOR AUTOMATION

When you know what types of issues you're likely to introduce and run into while performing development, you can start to look at automation.

Please see the next page for a table from xkcd that looks at how long you can work on making a routine task more efficient before you're spending more time than you save.

Be sure to take into account you probably have multiple people performing these tasks, and that automated tasks allow you to perform them more often than you otherwise might.

Something that's not included in the below chart is the idea that some tasks can be done incorrectly.

Automating a task, which is occasionally performed incorrectly, saves more than just the time saved.

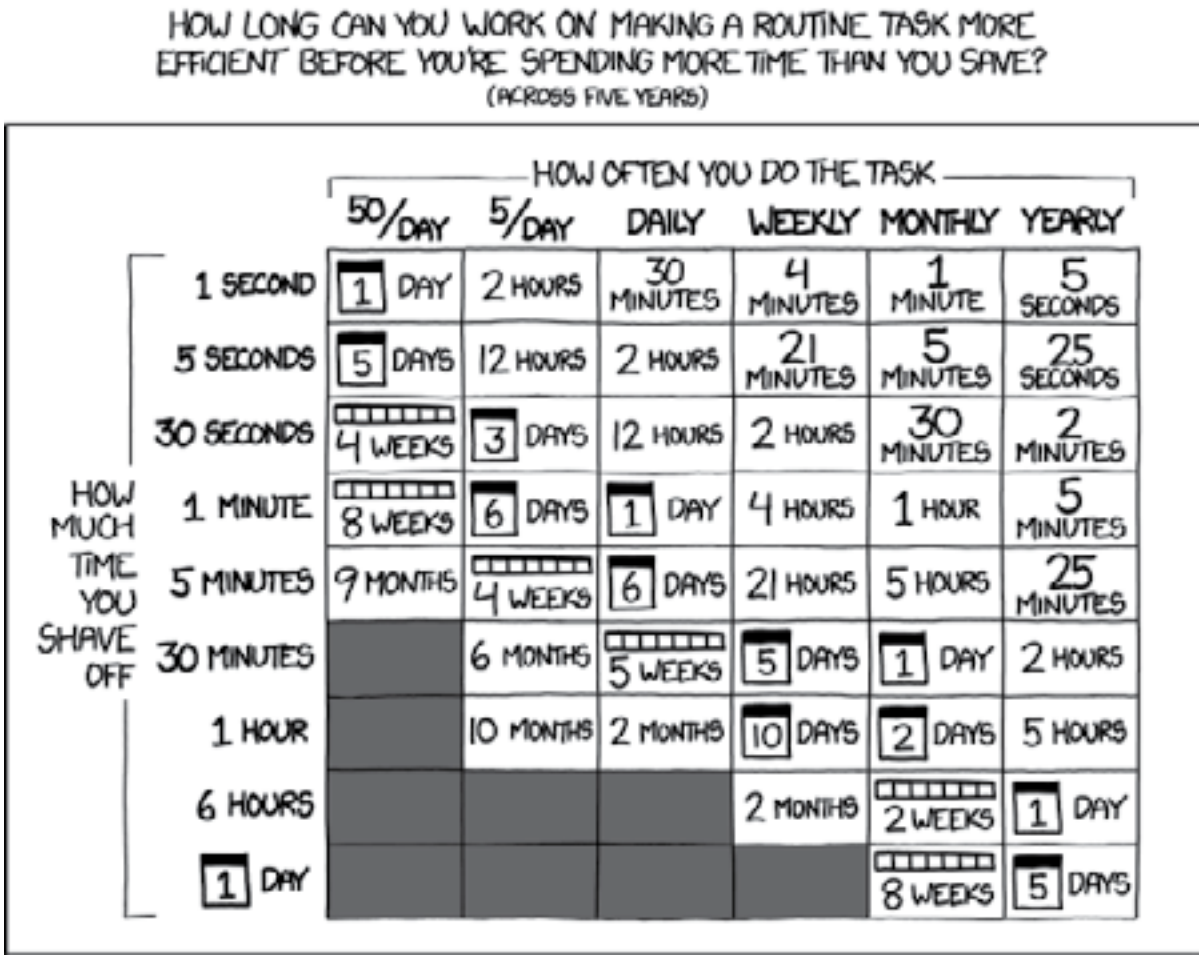
It provides a complete template for how the task needs to be performed, and effectively communicates this to other people.

When you automate a task, it no longer matters how much time it takes, so it is easy to scale up.

Say the task involves checking for syntax errors in some interpreted script, where a manual check might take 2 minutes per file. Most people are likely to perform this only for files that have been changed by them, and even then only when they are getting ready to update the code in source control.

On the other hand, if the checking process is automatic, it is much easier to validate the entire codebase, and it is possible to do so on every run of the software.

If it only makes sense to check updated files, this can be achieved automatically too (for instance via file system watches), which significantly reduces the risk of human error.



SOURCE: XKCD

AUTOMATION IS NOT A PANACEA

You cannot simply throw tools and processes at your problems and expect them to go away.

Software development is still a hugely complex endeavour. There are tasks that are simply not suited to automation. Things that are rare, and therefore should be verified manually, are prime candidates.

Only when you have advanced testing capabilities would you consider automating full-system scenario testing.

Certain companies such as Salesforce have 100,000 automated selenium tests that run on checkins — taking, in some cases, up to 12 hours.

It's important before investing in any significant level of testing to make sure that the thing you're testing is sensible, and to design the software (and test) to not be brittle.

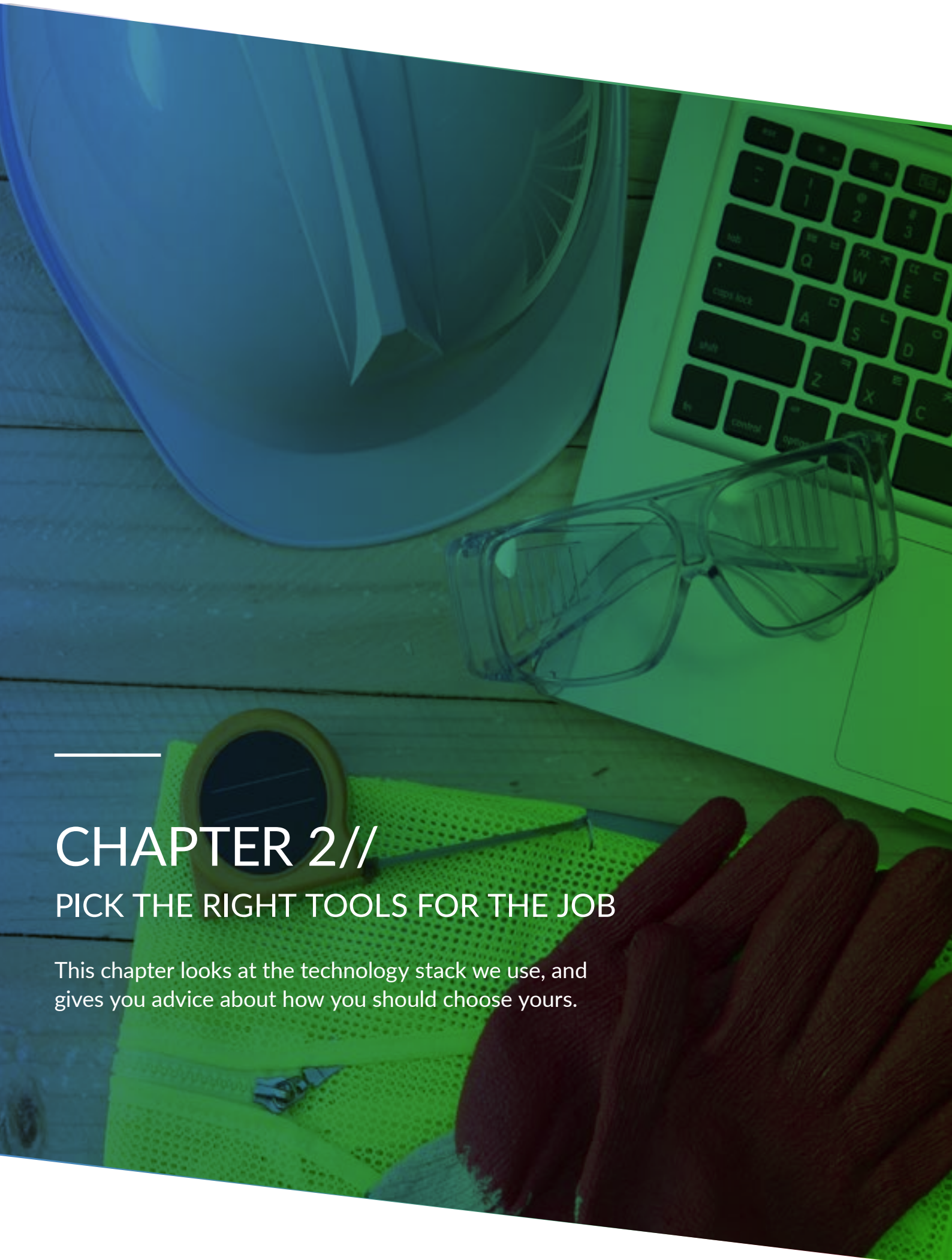
It only really makes sense to automate things that are stable, and only when you have advanced your development process and methodologies to a point where this is possible.

FAVOUR THE CONSOLE

GUI-based tools and processes are a huge hassle to automate. Wherever possible, try to use utilities and tools that can be run standalone and coded against.

Being only familiar with how to perform tasks within your IDE means that you are effectively insulated from the underlying technologies and how they work.

The tasks, and the way you perform tasks, goes 'against the grain' of the underlying technology. This is akin to how Microsoft tried to abstract away the web with ASP.NET webforms, and did so successfully, but at the expense of simplicity and efficiency.



CHAPTER 2//

PICK THE RIGHT TOOLS FOR THE JOB

This chapter looks at the technology stack we use, and gives you advice about how you should choose yours.

ONCE YOU KNOW WHAT SORT OF ISSUES YOU'RE LIKELY TO RUN INTO THROUGHOUT YOUR DEVELOPMENT PROCESS, IT'S POSSIBLE TO START TO LOOK AT PICKING OFF SOME OF THE LOW-HANGING FRUIT AUTOMATICALLY.

Even before you start looking at the defects you introduce, make sure that your broad development process is built on a solid foundation.

This process should have a few basic goals:

// Know the current state of the software.

// Be able to quickly provide an updated version of the software.

// Assist in planning current and future development

// Provide developers with a high level of productivity

// Mitigate common issues and problems

// Identify and resolve more in-depth issues as quickly as possible.

It's about knowing not only what you're making and how you intend to maintain it.

But also about knowing where you're at, where you need to go. And providing the framework to get there in the quickest and most effective manner.

As with a lot of things, what holds and works well for one team may not work for another. Find what works for your team and iteratively improve on that.

As a fundamental rule: the sooner an issue is identified, the faster, easier and cheaper it is to fix.

To ensure the highest quality software, make sure that you resolve problems at the earliest possible stage.

OUR TECHNOLOGY STACK

At F1 Solutions, one of our web-applications uses a relatively standard ASP MVC stack:

// C#

// ASP MVC + WebApi

// Knockout + Durandal

// LESS (+ some legacy SASS)

// An in-house ORM

// A ton of HTML templates

// JavaScript driving client-side functionality

// Microsoft Team Foundation Server for source control

// MSBuild for Continuous Integration

// Octopus for deployment

We picked these technologies to strike the ideal balance between our experience, functionality and development time (hence cost), as well as code and system quality.

The bulk of our server-side code is C#, which gives us strong typing, a compiler, a good IDE, and matches up well with our team's background and expertise.

ASP MVC + WebApi are well designed frameworks which are both extensible as well as predictable.

LESS and SASS allow us to structure our CSS styles in a way which is DRY (Don't Repeat Yourself), and provide some syntax checking on compilation/ transpilation.

We are stuck using HTML templates and JavaScript due to the nature of the web (but do what we can to keep these high quality).

TFS gives us solid source control and issue tracking capabilities and also serves our business users well. It also gives us a build system.

Octopus rounds out the stack by providing simple deployments our teams can use.

Of course, Your Mileage May Vary.

YOUR TECHNOLOGY STACK

The technology stack that you use should be driven by what you're hoping to achieve, your experience and also an underlying desire for building quality software.

Simply put, certain languages, platforms and technologies are easier to build high-quality systems with.

Certain languages, platforms and technologies make it easier to achieve certain tasks.

Certain languages, platforms and technologies align with your own areas of strength.

These considerations should all go together to help decide on your technology stack.

There's a whole other book worth of content that can be written on choosing a technology stack that pushes quality, but we find minimising state stored in components, injecting dependencies (constructor DI) and keeping methods small and focussed are useful approaches codewise.

So we pick technology stacks that allow us to do this with a minimum of fuss.

Wherever possible, try to collect metrics and as much information on errors as possible.

In an ASP.NET web application environment, this usually means some combination of ELMAH, emails and logging. Having this built into your design, from the beginning, reduces the time between developing something and knowing that there's a problem with it.

This means that you'll be able to fix problems faster.

YOUR DEVELOPMENT ENVIRONMENT

People have a lot of personal preferences when it comes to their development environment. What enables one person to be most productive may be completely unproductive for others.

TIP: Avoid forcing one development environment or toolchain on other developers. Improve development by looking at metrics, issues and outcomes — instead of trying to micromanage the development process itself.

Considering developer salaries, as well as the benefits that come from having highly productive staff, it naturally makes sense to ensure developers have access to the best tools that money can buy.

We find Visual Studio is a great tool for the C# side of things. Although, with the advent of OmniSharp, it's no longer the only player in town. But we also use a wide variety of tools throughout the team, including:

// Sublime

// Chrome workspaces

// Atom for editors

// Git + Git-TFS, TFS for source control

// IE

// Firefox

// Chrome for primary browser

// Node.js

// wWinless

// Visual Studio plugins for LESS and SASS compilation

// IIS, IIS express and occasionally express.js for a local http server

Overall, we use the things that work for us!

SOURCE CONTROL

Source control is essential to any software project. Setting up source control should be the first development task on any project.

If you don't have the budget to shell out for more expensive enterprise source control systems, the free alternatives such as. Git are equally capable.

TIP: There are no valid reasons not to use source control on any software project. If you don't have it, set it up right away.

ISSUE TRACKING

Like source control, issue tracking is a necessity. Without a good issue tracker, it's difficult to track whether they've been resolved.

Once they have been resolved, it's difficult to track whether they've been tested. Once they've been tested, it's difficult to tell whether they've been released.

The issue tracker acts as the confluence point for everyone in the team to be on the same page in terms of what work has been done and what has to be done.

Managing and resolving all defects found in an efficient way is an essential part of ensuring the highest quality codebase possible.

CONTINUOUS INTEGRATION

This is also one of the most important tools in a development team's toolbox.

Any time that you have multiple people working on a project, then continuous integration builds is a must.

The sooner you know if someone has done something which inadvertently makes the system unbuildable, the sooner you can fix it.

TIP: The third thing that you should do when working on a new project — after source control and issue tracking — is to get continuous integration up and running. It's the most important basic indicator of the health of a codebase. It also provides hook-points for other automated quality tools.

Consider making a build containing all compilable source code to be the minimum success criteria for continuous integration.

This will find syntax errors and misreferences within compiled code in the source. Of course, developers should be checking things before they prepare source code commits, however all sorts of errors, primarily human, can break the build.

Once this is done, consider implementing continuous deployment. This deployment will help ensure there are no logical errors introduced into the build and deployment processes, and will also help uncover errors in data migrations.

REFACTORING

Automated refactoring tools are one of the most important recent code quality innovations.

Without this tooling, refactoring is slow, complex and error prone. For C#, Visual Studio ships with a number of basic refactoring options, which, when combined with third party tools, improve overall developer productivity significantly. (We use ReSharper; other options include CodeRush.)

Microsoft has noted the need for automated refactoring, moving over to the Roslyn compiler and significantly bulking up code analysis and refactoring options in Visual Studio 2015.

TIP: If you're developing C# or VB.NET but you aren't using VS 2015, and don't have it already, get yourself a copy of ReSharper. Your codebase's quality, as well as other developers, will be glad you did.

TRANSPILATION

Tasks like LESS file compilation are things that need to be performed whenever a developer changes a file. There are three approaches you can take for these sorts of tasks:

// Use a tool built into your IDE that handles it all for you,for example Web Essentials in Visual Studio.

// Use an external standalone tool that watches and transpiles your files for you,for example WinLess.

// Manage your transpilation from a console.

We recommend you favour the last option, as this most readily lends itself to automation and extension.

In the case of less, using the less compiler through a node.js console is simply:

```
lessc graphs.less > graphs.css
```

This type of behaviour is far less 'magic' and far more understandable than configuring something in an external application, even an IDE. Note: in the new versions of Visual Studio, the IDE provides a means to execute these types of tasks, which goes through the console.

TIP: If you're transpiling code, make it an action that occurs when you build the software. Don't check it into source control. This will remove the possibility that someone updates the source file but not the output.

PACKAGE MANAGEMENT

Package management is something that, until recently, was completely ignored in the .NET world, but utilised heavily elsewhere. Historically, Microsoft saw the Global Assembly Cache (GAC) as the solution to DLL Hell, where developers would run into problems with users not having the correct version of dependencies installed, along with some other versioning issues.

Unfortunately, while the GAC did provide some relief from these problems, it introduced others.

Shortly after realising this, NuGet entered the scene. NuGet is a centralised repository of versioned packages that can be installed into to your .NET projects.

NuGet improved the situation significantly, but revealed other problems with the way .NET projects were historically structured (mainly around things like installing and updating web application dependencies).

Many of these limitations and problems are being addressed with asp.net vnext, the next generation web application technology from Microsoft.

Instead, what is most forward looking is to use two package managers from the JavaScript world: npm and bower.

NPM is a package manager that is useful for development with JavaScript.

Bower is a package manager that is useful for getting packages built by others into your web applications. Both can be executed through node.js.

Both have some level of support built into Visual Studio 2015.

TIP: There is a lot more to cover on to effectively use package management and build tool which we will save for another time. Package managers try to, and to an extent do, handle many issues with legacy components and external components changing. They give you the ability to track and more easily perform updates to your external components.

BUILD TOOLS

When developing code which is not compiled at runtime, it quickly becomes burdensome to manually perform the tasks that need to be done when a file changes (for instance transpiling LESS files to CSS). For this, the *JavaScript World* has a number of good answers. Node.js + any number of build tools, such as grunt and gulp, are a good solution for this.

Most of our expertise lies with using grunt, where you create a gruntfile. This is a declaration of a number of tasks built up of component plugins that can run on a command ("grunt task-name").

One very useful task is a watch, such as grunt-contrib-watch, which allows node to sit and wait for changes to one or more files.

This can be combined with things like lessc, via the grunt-contrib-less, to compile less files to CSS whenever they change.

A NOTE ON WARNINGS

Warnings, such as those given to you by your compiler, IDE or tools, are useful and great. Two quick notes on warnings though:

1. Warnings may not themselves indicate something is necessarily a problem. However they might be, and having a large number of warnings provides additional noise within a codebase.

   If you have no errors and add one, it's very noticeable; however, if you already have 500 warnings, you are less likely to notice a 501st being added. And that 501st warning might be one which is a real concern.

TIP: Treat non-stylistic warnings as errors. If there's a way that you can enforce this in your IDE/Build system, switch it on.

2. Most large codebases will have a number of areas where the code doesn't comply with z standards and causes warnings. New codebases most likely won't. When you first start developing functionality, it's easy to focus on getting stuff done and ignore quality.

However, when you finally do get a chance to spend time looking at code quality, it's a much larger and more daunting task. It's best to start and build on a solid foundation by thinking about your automated tools upfront and enabling linting, code analysis and other quality tools from the get-go. The following sections will cover some key considerations.

TIP: Code quality is difficult to add across-the-board as an afterthought. Do it from the beginning.

COVER OFF THE BASICS

Once you've got source control, issue tracking and continuous integration builds down, it's time to start looking at improving the quality of the code itself. There are a few across-the-board basics which are reasonably simple to get coverage on.

SPELLING ERRORS

Code-aware spellchecking is a must if you embed your user-facing text in code. There are plugins for essentially every editor/language for this.

Spell Checker, ReSpeller and Softario are a few examples of plugins that handle C#. Even if you don't embed your user-facing text in code, these tools are still invaluable in ensuring variable and method names are correctly spelled.

Structuring your code in such a way that user-facing strings are kept separate is a good and useful best-practice. Doing this lays the bedrock for product internationalisation. It also segregates this information in its own place, making it easy to rename product concepts. Another benefit is it allows for non-technical users to check and provide the text that's used is spelled correctly.

Some options for this include resx files, if you write Microsoft, a database localisation table, or some sort of client-side resource, such as JavaScript/JSON).

CLIENT INCOMPATIBILITIES

If you're developing for the web, then you're in luck. Twice as much if you are primarily targeting desktop systems.

Tools such as Browserstack and Sauce Labs let you perform manual smoke testing via the internet. Ghostlab is a nice local tool for locally testing multiple browsers (though, we've found it not to work well with our applications).

For testing against older versions of Internet Explorer, Microsoft has some useful test VMs available over at modern.IE

If you're testing windows applications, or testing your server components, you'll likely wind up needing to create and maintain a large number of VMs to cover off important configurations.

If it's your first time doing this — pick a good naming scheme, and be sure to disable automatic updates!

Where possible, try to automate the golden path of your testing. Tools such as Microsoft Test Manager have proven their value to us in doing this.

TIP: Invest time in making your software easy for automated testing tools to use, and be sure to involve your testers in this process. Having confidence in the basic functionality of your software's releases frees up test time to focus on more important things.

SYNTAX ERRORS AND MISREFERENCES IN INTERPRETED CODE

When using interpreted code or markup, it is often possible to write code which appears to work but has syntactical or reference errors. Some examples of interpreted code or markup include JavaScript in the browser, PHP on most servers, HTML, various templating and scripting languages.

There are two different paths you can generally take:

1. Figure out how to write something compilable
2. Figure out how to validate your code. In the context of JavaScript, these options roughly boil down to writing your code in something else and compiling it (e.g. CoffeeScript or TypeScript) or running it through a linter (e.g. JSLint, JSHint, ESLint, Flow).

TIP: If your codebase contains JavaScript, get linting into your builds ASAP. Pay attention to the warnings.

Either of these approaches should identify and eliminate basic errors with syntax. Compilation systems such as those provided with typescript will significantly reduce misreferences in code. And linters should reduce misreferences within each code unit, for example source file. Breaking code into small modules from the beginning makes linting much easier.

ESLint is an interesting project in that unlike JSLint/JSHint, it allows for custom rules to be created and added easily.

This gives you the ability to add/enforce your own requirements (for example if you have deprecated JavaScript methods; want a particular method's return value to be used; or want to enforce a particular style) which is valuable.

STATIC ANALYSIS TOOLS

In the .NET world, there are various static analysis tools.

Two of the more well-known are Code Analysis, which is a replacement for an earlier Microsoft tool called FxCop, and NDepend.

In general, static analysis tools have diverse goals.

But, generally, the concept is that the code or compiled outputs of the code are loaded into the tool and analysed against a set of design and structure rules to generate a list of areas where the guidelines are not followed.

Like with warnings, these tools often give a lot of false positives, but they also catch legitimate problems and add a lot of value when used correctly.

Where the balance of "correct" lies for your team depends on the nature of the software you're making.

Similarly, in other languages, static analysis tools look for 'risky' language choices, such as assignment within if conditionals.

A list of static analysis tools for other languages is available at this link.

NDepend has an interesting feature called CQLinq, which allows you to manually write queries against the codebase.

We'd definitely recommend checking this feature out.

It allows for analysis like finding methods with names that are too long to type, methods that are unused across a set of projects (even when public), fields that don't follow naming rules, usage of specific types and so on.

In short, it's very powerful stuff for finding areas of your codebase where there could be problems.

CODE STYLE TOOLS

Various tools exist to enforce code style. In most cases these are built into other code quality tools (ReSharper and JSLint both have style rules).

Some standalone tools also exist, for example Microsoft's StyleCop.

Note: the styles that are shipped with these tools may not all work with you. They might even give you a lot of grief when you have an existing codebase that does not match the style.

OTHERS

PowerShell is something we make significant use of but isn't necessarily directly tied to our quality processes, except in automating other activities.

Unit testing and other automated testing is something that we also do but that deserves its own section.

Same with metrics collection – if possible have feature usage and errors reported to you so that you can make product decisions with information backing up your assumptions and beliefs.

Having errors reported makes it faster to uncover defects.

There are a number of other tools which we use for specific situations but unfortunately we don't have the time or space to cover them in this book.

Some good general advice is to look at the problems that you run into – the places where you have recurring problems with quality – and start to look for tools that can help improve on them.

IN SUMMARY

There are a wide variety of very useful tools that can really help improve code quality. We recommend a number of them, configured in such a way that warnings are shown as errors:

// A good IDE picks up compilation issues quickly, show warnings and errors during compilation, quickly navigate around source files, get syntax highlighting. We use Visual Studio.

// Source control sees what's changed, provides a canonical representation of the code, handles problems with multiple developers working on components at the same time. Git is free and works well.

// An effective technology stack — we find C#/ ASP.NET MVC/ASP.NET WebAPI/JavaScript/ LESS/HTML along with various frameworks works well for most uses — identifies tools and technologies that let you write high quality code by default, and pick up errors as soon as possible (i.e. favour compiled languages).

// A good issue tracker keeps everyone on the same page and improves the speed at which you can find and remove quality issues from your codebase. We use TFS but GitHub is a solid workflow, too.

// Continuous integration gives you an additional layer of protection against human error. It also helps ensure your system can be built and shipped at any time. Additionally, it provides a platform to automate other quality tasks on some centralised infrastructure.

// Automated refactoring tools allow you to avoid adding defects when you make changes. We like ReSharper.

// Transpilation allows you to write code in a less error-prone language. You can automate this process. We like lessc via node.

// Package management reduces the chance of external changes breaking your software and lets you choose when to update components. Examples include NuGet, NPM and Bower.

// Build tools let anyone make replicable builds and centralise the dependencies of the application, making the build Canonical.

// Code aware spelling tools and methodologies prevent simple typos and other things that make the software look amateurish. =They also prevent typos in serverside code, which may make members and methods less discoverable.

// Testing tools make it faster to test your software leaving more time to identify and resolve complex issues.

// Linters give you basic assurance your interpreted code is at least partially correct.

// Static analysis tools provide protection from unsafe patterns within compiled code, direct developers away from ambiguous calls and otherwise identify problems.

// Code style tools help keep code consistent across the codebase, making it easier to read.

// Use various other tools, as required.

CHAPTER 3//

DECIDE ON YOUR STYLE & STICK TO IT

A defined code style is something that is important to have, to document and to enforce. This chapter looks at how to decide on your style and stick to it.

A DEFINED CODE STYLE IS SOMETHING THAT IS IMPORTANT TO HAVE, TO DOCUMENT, AND TO ENFORCE. WHAT DEFINING A STYLE DOES IS ENSURES THAT ALL OF THE CODE THAT'S WRITTEN WITHIN A PROJECT LOOKS CONSISTENT, AVOIDS AMBIGUOUS PATTERNS AND HAS THE ULTIMATE GOAL OF MAKING THE SOURCE CODE EASIER TO READ (AND HENCE UNDERSTAND).

TIP: Something that a lot of people don't realise however is that unless your style is objectively bad, it doesn't really matter what the style is. What's more important is that it's consistent. As such, when you are working on a project that has its own style conventions, you should always use those for consistency.

A while back at F1 Solutions, we attempted to have one unified style guide. Any code that we wrote — be it C#, JavaScript, shell scripts — would try to follow this.

We found, practically, it was a terrible idea.

Our JavaScript would match up nicely with our serverside code. However, whenever we called any external libraries, these used a different capitalisation scheme, which made the code inconsistent and difficult to write.

Developers had to remember for external library methods there was one naming scheme and for internal library methods there was another.

While this doesn't seem like an overly major issue on the surface, it did increase friction and make development in non-.NET languages slower.

We have since learned from that mistake.

Now, we maintain a separate set of styles for .NET and JavaScript, and delegate other styles to however the authoritative sources for information on those languages.

The styles that we use, like with many development organisations, are based on publically available documents describing style.

For C#, we use a derivative of the *Microsoft C# Coding Conventions* and have our automated tools configured with the rules that match what we use. For JavaScript, our conventions are more loosely based on Douglas Crockford's *Code Conventions for the JavaScript Programming Language.*

These are the prevalent styles for each language that we use, and find that external libraries tend to follow them at the API design level. Our modifications are largely around whitespace, comments and edge cases not covered.

AUTOMATING STYLE

Style is something that can be checked automatically.

For C# tools, such as ReSharper or StyleCop, perform this type of checking. For JavaScript, there's JSHint/JSLint/ESLint. For LESS, there's RECESS by Twitter, which is, in our opinion, a little too strict.

These tools can all be hooked into a build process to make sure developers are following the style guidelines and to provide feedback to developers when they don't.

We'd recommend the automation of style checking. However, the tools typically cares about things we don't, so turning off some of the options is often useful.

On an existing codebase, it's quite likely there will be a large number of warnings, so this is something that may also determine which rules are enabled where.

Note: style tools are one exception to the "treat warnings as errors" rule. Some style problems could be treated as errors, but many are okay, even though the rules that you use should be practical for you.

What's important is knowing when new warnings are introduced.

As such, we leave our tooling in ReSharper on with warnings so we can see them while writing code, but we don't have these displayed as errors.

TIP: Configure style checking tools in your development environment. Switch off the warnings that aren't relevant to you. If you're working on a new project, try not to introduce problems. If you're working on an old project just fix the areas that you touch. Don't treat these warnings as errors.

STYLE ISN'T JUST CODE, IT'S STRUCTURE TOO

Style comes down to expectations.

When developers understand how something should work, they are annoyed when they find an exception to the rule or something that they expect should work but doesn't.

That's why it is essential to have consistency in design throughout the codebase.

To this end, in our dependency injection system, we never have an interface that can be seen from two components but can only be created in one of them.

We avoid storing state within objects. We keep methods simple and have a fair amount of plumbing, which is occasionally boilerplate code, between them.

We find that while it takes time and effort to implement the boilerplate, it saves time in the long run because components work in the same way and we can more easily reason about the separation of concerns within the system.

In fact, this is much of the rationale for using software design patterns in code — they are a library of named concepts that developers understand.

It also helps that they solve problems well. When properly named, these components make it clear to the reader in just a few seconds how multiple components interact and work.

That's the whole point of having consistency within your codebase. Being able to understand how something works in seconds rather than minutes makes fixes faster. It also makes identifying problems easier and makes fixing them more straightforward.

To ensure the structure is well kept, we create and maintain an overarching system design document which captures technical information about how the projects are structured, how they interact, the high level technology choices along with any important notes that a new developer may want or need to know.

TIP: Maintain a document which covers high level architectural decisions, along with associated documents that describe how various cross-cutting concerns should be implemented. This can be useful to senior developers and other staff, but also to explain how things should be done to juniors. These documents should be distinct from system requirements.

Some other structural considerations can be encoded into this document too – targets for method length, complexity, coupling, storage of state and other code metrics which have a correlation to quality can be defined here.

STYLE IS USER EXPERIENCE AS WELL

What is true of developers and code structure is also true of end users and the software's interface and operation.

If you have a pattern that's used throughout an interface, the exceptions to these rules are sure to frustrate users who are surprised when they are not met in individual places.

Say, for example that your application automatically saves when the user navigates from every application screen, except one.

Without a visual cue, users will visit the page and expect their changes to be automatically saved. When they aren't, they'll be confused and then annoyed.

TIP: Define a style guide – a document that covers your basic interactions, elements that should be used for them, brand colours and fonts. Make sure that new components use these styles and underlying structures.

It's likely that you don't need a formalised document as big as *Yelp's Style Guide* or *Github's Style Guide*.

But these can be easily slimmed down for usage within your own products. It is really not possible to overestimate the effect that a good UX style guide can have on the consistency of a product.

Building on top of consistent and succinct styles in a structured manner ensures:

1. Fewer graphical changes —fixes as a result of someone asking why something works in two different ways in the system
2. Reduces bugs as they would be duplicated across all instances of the component and therefore they are noticed far sooner than they otherwise might have been.

A nice side effect is that it becomes more easy to safely refactor the component's instances.

This is a major benefit considering refactoring HTML templates is notoriously difficult at the best of times.

IN SUMMARY

A code style or set of code styles allows developers to more quickly understand what is happening within a single method.

Many of the code-level style rules can be enforced by tools. However, unless you want a spotless codebase, we'd recommend just resolving the warnings provided by these tools, wherever relevant and possible, and not treating them as errors.

Style is also broader than just where you put braces and whether you wrap your single-line ifs. It goes down to the structure that you want a system to have.

In addition to the design patterns the system utilises, including broad as well as Gang of Four, the architectural style goes down to the level of how data is transmitted and encoded. It documents how big methods should be, how their state should be stored and anything else that is relevant to developers when they are looking at adding new functionality to the system.

To the user, style means something else as well: the way that the application looks and feels.

All three of these are difficult to communicate directly with other developers and non-engineering staff.

We recommend putting these rules and structures down into documents so everyone's on the same page and there is a canonical representation of how you would expect components to work.

The name of the game is keeping things consistent.

We often can't rely on code itself to speak to developers, analysts and others, and wherever things are inconsistent you are likely to find complexity and bugs.

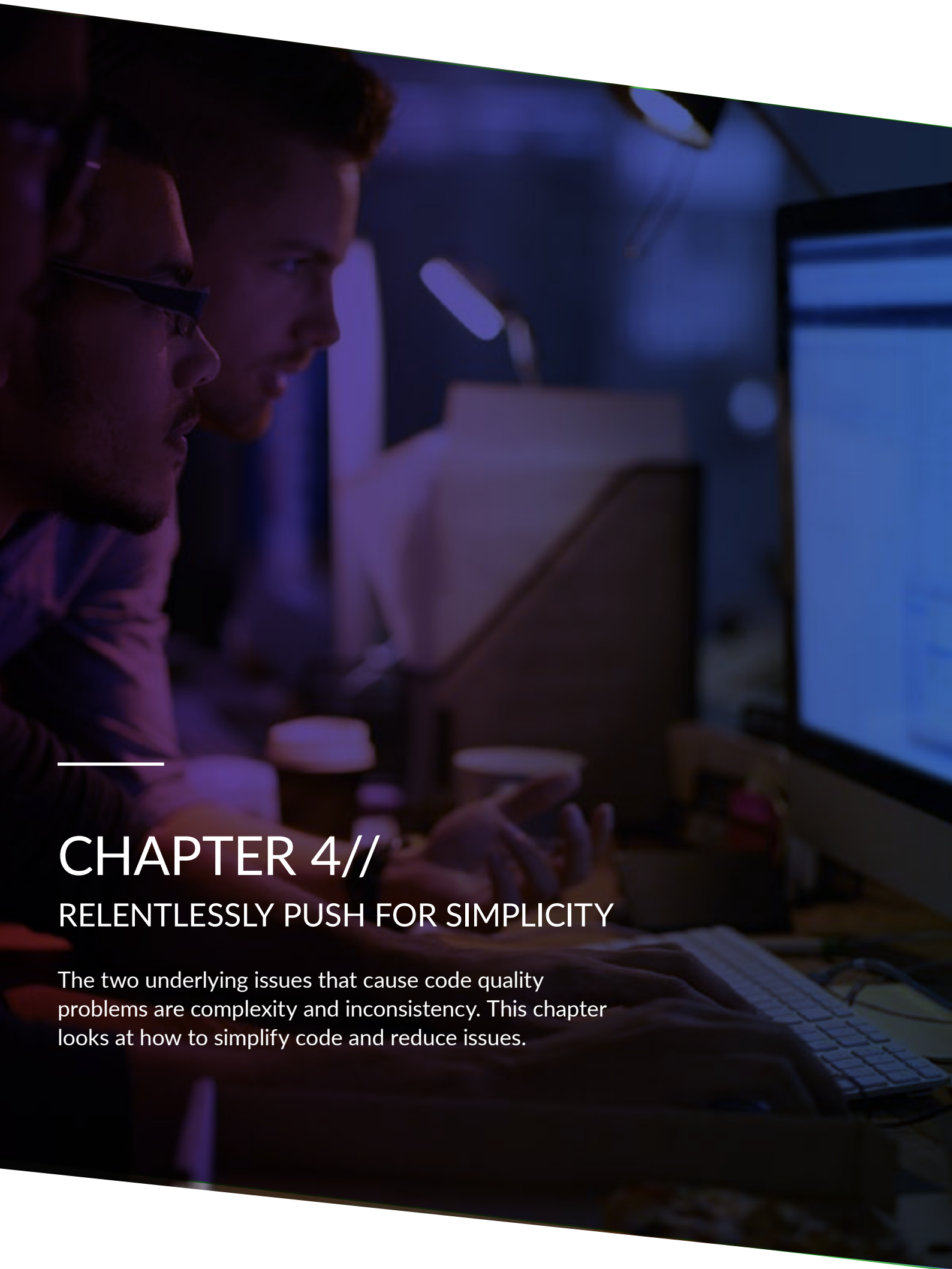
Note: a side effect of this is not making hacky fixes.

If something is broken, any fix should be applied across the board and not on a piecemeal basis.

Of course, this doesn't mean fixing anything that isn't broken but it means if you are having trouble with one instance of a component, you should find out why this instance is broken and apply a fix that detects that condition and rectifies it.

This may sound like a simple concept, as it is simply Don't Repeat Yourself, but in practice it's often overlooked.

Having this set of rules and consistency keeps things working the same across the board.



CHAPTER 4//

RELENTLESSLY PUSH FOR SIMPLICITY

The two underlying issues that cause code quality problems are complexity and inconsistency. This chapter looks at how to simplify code and reduce issues.

THE TWO UNDERLYING ISSUES THAT CAUSE CODE QUALITY PROBLEMS ARE COMPLEXITY AND INCONSISTENCY. INCONSISTENCY CAN COME FROM STYLE, BUT COMPLEXITY COMES FROM A NATURAL TENDENCY FOR SOFTWARE TO GROW AND BECOME BLOATED OVER TIME. THIS IS SOMEWHAT RELATED TO TECHNICAL DEBT.

Where you have complexity, you are likely to find defects and other quality problems. Where you have inconsistency, you are less likely to notice problems. In both cases you are more likely to have a harder time trying to fix the underlying problems than with an issue in simple and consistent software.

Complexity takes many forms in a codebase, from lines of code, method calls from a function, the number of external dependencies used by a class, all the way down to an unidentifiable feeling that something's not really that clear from reading a class.

Complexity also takes many forms in an application.

Sometimes this complexity is necessary for the successful operation of the system. Other times it is unnecessary and results in:

// Duplicate functionality

// Actions that require multiple clicks instead of just one

// Additional screens that offer little value

// Graphics that add no additional context

// Processes that could be far simpler.

Regardless of the cause of the complexity, it is important to remain vigilant and ask if every individual feature is used, and therefore necessary.

Similarly it is important to try to combine features. While it may be easy to add a new feature, modifying an existing one to make it more powerful without adding cruft is a different challenge.

TIP: When developers work closely with analysts, the solutions that are proposed tend to be the best from a simplicity viewpoint. An analyst should understand what the business wants, however a developer should ask if each of the "parts" of a feature is actually useful.

DON'T MANAGE BY KPI

You cannot manage your codebase by KPI.

There are useful metrics that can come out of static analysis tools, such as NDepend. For example, you can identify potential problem areas; although it is worth mentioning some features are naturally more complex than others.

Some components may need to maintain state.

The same can be said of number of defects from a particular subsystem. Defects do not necessarily mean an area needs to be simplified, even though defects may be an indicator of problem areas.

Instead, simplicity is something that needs to be sought manually. Skilled engineers are really the best, and only way, to ensure a codebase contains code which is simple.

USE 3RD PARTY COMPONENTS

A great way to manage complexity within your codebase is to try to move a large amount of suitable complexity to a third party.

Of course, you must ensure it's something for which a trustworthy third party exists, and their solution is suitable for your needs.

For every non-differentiating and non-core feature of your product, you should ask whether some COTS would be better than what you've already got.

Failure to do so diverts valuable development resources to solving problems that are already solved. It also dilutes your application and team, and usually winds up costing more than it otherwise would.

For instance, if you're working on a staff time tracking system, it may be important to generate PDF reports.

Actually figuring out how to build PDFs from scratch might be interesting. But unless there's some direct value in doing so, you should look at COTS solutions for PDFs.

When you look at how many hours would be involved in scoping, designing, developing and testing your own PDF generation system, almost any amount for a COTS product starts to make sense.

On the other hand, if you're working on a time tracking system and want to develop some functionality that monitors the active window on the user's machine, there is likely no COTS product that fills this need, so it should be developed and maintained in-house.

There are two extremes to avoid in this regard:

1. Not Invented Here (NIH)
2. Proudly Found Elsewhere (PFE)

Not Invented Here is when third party solutions are rejected outright only because the source code doesn't belong to the developer.

This is a problem because there are many utilities and components that are either free or inexpensive, compared to in-house development.

Not to mention these would have gone through much more development and testing than you will be able to fit into your current budget. NIH basically ignores these benefits.

Proudly Found Elsewhere, by contrast, is always favouring external solutions. This usually stems from organisations not trusting their own staff or wanting to have someone external to blame for the failure of a component or project.

TIP: Staying in between these two extremes is essential to delivering cost effective software that is of high quality. You need to critically evaluate external options, but at the same time not be afraid to build your own components if they are core to your application, don't exist yet or are unsuitable for whatever reason. If an external option is suitable, you should seriously consider using it, as the ongoing costs for developing a component to a high level of quality are usually much higher than just purchasing a license.

NEVER ROLL YOUR OWN CRYPTO

"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."

- A quote from Schneier's law, which is named after Bruce Schneier who is a renowned security researcher. This quote captures a number of issues and problems facing computer scientists, from developer hubris to Not Invented Here syndrome to the true complexity of creating working security.

Real experience with cryptanalysis is required to successfully implement cryptographic components.

The challenge isn't creating something you cannot break or cannot be easily broken by others; it's creating something that holds up to an extended attack by determined experts.

If you do not have the appropriate security background, leave it to someone who does.

TIP: Never roll your own crypto. Just don't.

REMOVE DEPRECATED AND UNUSED CODE

When code is no longer called from within the system, remove it from the codebase. If you need that code again, look for it in source control.

Judiciously removing anything that's no longer necessary keeps the codebase lean and focussed. This makes it easier to find what's in use at present, allowing developers to find and fix problems more easily.

KILL UNUSED FEATURES

This is always a difficult call to make, but features that are no longer used, even if they are exposed to users, should be removed wherever possible.

If this is not possible, the feature should be rolled into whatever replaced or made it obsolete. Unused features are like deprecated and unused code – they bloat the codebase and as time goes on make it harder to make changes.

Of course, actually understanding which features are used and which are not used is difficult.

Wherever possible, you should be basing your decisions on hard numeric metrics of user behaviour. Alternatively, remove the feature from the default UI and hide it behind an option. Measure user interest in that feature; if there is little or none, remove it from the system completely.

BUILD ON A FEATURE, DON'T CREATE AN ALTERNATIVE

This is a rephrasing of the "Kill unused features" point above.

If you are going to provide an alternate way of doing things, try to figure out how to combine the two — how to modify the existing functionality to do whatever needs to be added.

Sometimes modifications cause code issues. But the ideal in the trade-off between bulk of code and risks of changing behaviour often lies closer to keeping existing features but extending them.

AVOID UNNECESSARY AND USELESS ANIMATIONS

Our brains are naturally drawn to animation, so it should be used sparingly and strategically.

One problem with many computer systems is animation is used far too often to present information simply for the "wow factor".

What this does is distract and confuse the user. It also complicates the codebase. In our experience, animations also tend to be a source of defects.

Instead, animations should be used for a purpose. An example might be to draw the user's eyes to something important like contextual information or important data. Even then, animation should be simple and subtle.

Taking this approach with web-applications allows you to use CSS animations in lieu of more complex JavaScript-based ones.

We find that CSS is more succinct, simpler and, compared to home-grown JavaScript solutions, has fewer issues.

CHAPTER 5//
FOCUS ON CODE COMPREHENSIBILITY

Good code doesn't just work; it communicates its purpose to the reader. This chapter gives tips and advice to help you write code that others understand.

"THE RATIO OF TIME SPENT READING (CODE) VERSUS WRITING IS WELL OVER 10 TO 1... (THEREFORE) MAKING IT EASY TO READ MAKES IT EASIER TO WRITE."

- ROBERT C MARTIN, CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP

Good code doesn't just work; it communicates its purpose to the reader.

Similarly, a developer must understand how the code works in order to effectively and correctly make changes.

This is particularly important because the majority of a developer's time involves reading code to understand how it fits together in order to rationalise what impact changes will have on the system.

Any attempt to improve code quality and reduce the frequency of defects should be built on the concept of keeping code readable.

This chapter broaches the broad concept of code clarity, or writing understandable code.

In defining what we mean by understandable, we can boil it down to a simple question: If you were to give some code to an average developer who is familiar with the language, can he or she understand its intent — what it does — and its mechanism — how it does it — by simply reading it? When the answer to this question is "no" the code is not clear and there is room for improvement.

Note: it is often said code should be self-documenting. While this is sometimes offered as an excuse for not writing comments, it is nevertheless a claim with merit. There are few statements you can make about code that is easily understood.

Code that is easy to understand:

// Is written in such a way it communicates its intent to the average programmer reading it.

// Does not employ unnecessarily complex or obscure techniques.

// Dependencies are easily observed.

// Requires few comments, but is clearly commented where necessary.

// Employs consistent naming and formatting conventions

Roughly speaking, this understanding, and the broad concept of code boils down to ensuring that there is consistency across the system, and that the code itself is predictable and as simple as possible.

Comprehensible code involves:

// Structural consistency

// Stylistic consistency

// Keeping functional complexity as low as possible

// Keeping components loosely coupled

// Be explicit with flow

// Comment where necessary

// Be explicit and consistent with naming

// Be explicit with preconditions and postconditions

// Avoiding complex and oft-misunderstood language features

// Ensuring methods are appropriately sized

// Keeping scope depth low

This book has already touched on the first three items, which are important but by no means enough to keep code in a state that it is easily readable.

WHY ALL THIS MATTERS

Bad code is a business liability.

It is a given that the code we write must, when compiled or interpreted as a program, do what it was designed to do.

That is its principal reason for existing. And that is usually foremost on the mind of the developer writing it.

What is often forgotten, though, is its secondary but equally important role: to efficiently communicate its intent to a human reader.

When our code fails in this secondary purpose, it is no longer good code — no matter how well it works.

Rather than the asset it should be, it has now become a liability.

BUY NOW, PAY LATER

Writing easy-to-understand code comes at a cost. It takes longer.

But writing hard-to-understand code also comes at a cost. Sure, it might be cheaper in the short term but hard-to-understand code often results in technical debt, which is a latent cost to its owner.

Technical Debt manifests in several ways:

// Debugging or enhancing it takes longer, as developers must spend time deciphering it before they can modify it. This is a repeating cost; every developer who reads the code must expend some effort understanding it. In this way, a few minutes of creation can lead to hours of unnecessary time spent. All of this time would have been saved if the code was written clearly to start with.

// Unclear code gives rise to more bugs, because developers making changes to it are unlikely to fully understand it, leading to mistakes. And more bugs, of course, mean more expense.

// It is less likely that subtle bugs in cryptic code will be noticed by developers; thus increasing the likelihood of bugs reaching production. This lowers the quality of the product, which ultimately can affect customer confidence.

BAD BEGETS BAD

Worst of all, bad code is a disease that rarely remains isolated.

Just as financial debt grows through interest over time, bad code tends to proliferate as time passes by.

There are various ways in which this happens:

// When an enhancement leads to code being added to a poorly-written module, it invariably becomes a bigger poorly-written module. It is difficult to build a strong structure on a weak foundation without rebuilding the foundation first.

// Developers will often, with the best of intentions, copy existing code when developing a new module which is very similar to an existing one. This is often the only choice given a constrained schedule. But it has the effect of reinforcing the bad code, which has now become a local pattern within the project.

// Junior developers who are learning by example will often copy existing patterns without realising that they are bad patterns. The same goes for some senior developers who may be working in a language new to them.

// Eventually, when so much of a codebase is of a low quality, the problem seems too big to fix and developers become apathetic and simply stop trying to fix it.

This compounding effect continues as long as the problem goes uncorrected.

Ultimately, the quality of a product's codebase can decide the fate of that product.

A complex system composed of largely unintelligible code eventually becomes too costly to maintain, and is abandoned or redeveloped at great cost.

In contrast, a well-written system is likely to go on being maintained and enhanced for longer, extending its useful life as an asset to the business.

IN SUMMARY

If any part of your livelihood depends on the software you write, bad code is costing you.

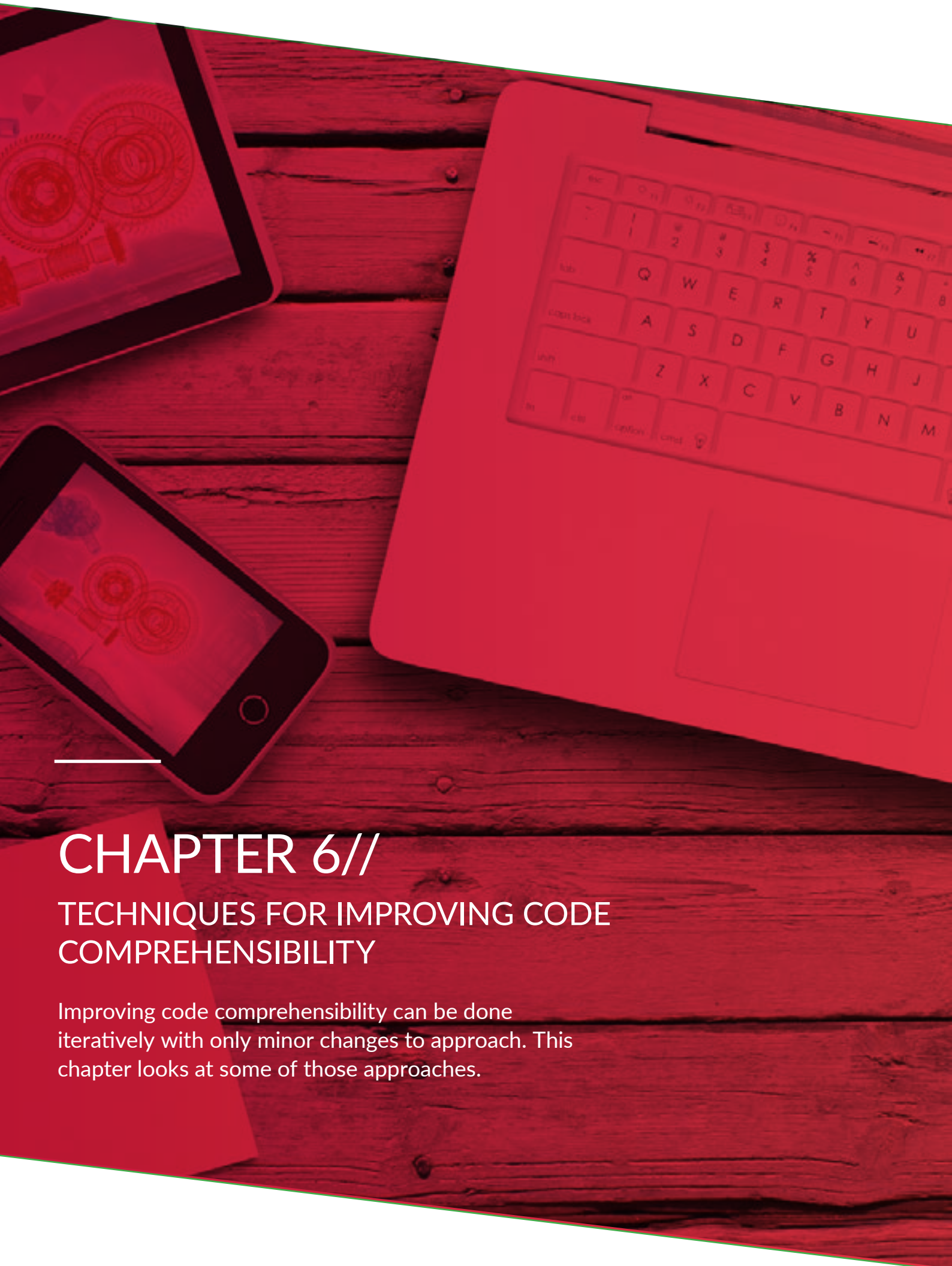
It's costing you valuable developer hours in deciphering it and debugging it. It affects customer confidence thanks to a higher number of production issues. And let's not forget that it affects revenue due to longer release cycles.

Writing clean, understandable code is not difficult, but it requires some discipline.

It seems to be a natural law of the universe that good habits are more difficult to hold onto than bad ones, and so we find it an up-hill battle to keep bad code from creeping into our systems.

The chief enemy of quality code may be the deadline. Those of us beholden to the demands of customers are rarely afforded the luxury of taking the long way around. But the mistake is to think of it as a luxury, when it should be considered a necessity. The key take-away here is that whatever may be the present cost of addressing bad coding practices, the future cost of ignoring them is guaranteed to be greater.

In the following chapter, we explore some straightforward approaches to minimising complexity and incomprehensibility in code.



CHAPTER 6//

TECHNIQUES FOR IMPROVING CODE COMPREHENSIBILITY

Improving code comprehensibility can be done iteratively with only minor changes to approach. This chapter looks at some of those approaches.

FORTUNATELY, IMPROVING CODE COMPREHENSIBILITY CAN BE DONE ITERATIVELY WITH ONLY MINOR CHANGES TO APPROACH. WHEN COMBINED WITH REFACTORING TOOLS, CODE CAN BE IMPROVED, ONE METHOD AT A TIME.

LAW OF DEMETER

The Law of Demeter (or Principle of least knowledge) is a design guideline which states that each unit should only have knowledge about the units that are closely related to its functionality. The underlying rationale is that any piece of the system should only know about itself and the bits of data that it relies on – in short, any component should not need to "reach through" another piece of data or functionality that it doesn't need to know about to perform its task.

For example, imagine an online shopping system which automatically ships parts from a warehouse as close to the user as possible. A user orders some products and enters their address when checking out. This address is stored against the user record. The product order lines (which reference the product) are stored against an order record, which is associated with the user.

Imagine a function `Warehouse GetClosestWarehouseWithProduct(ProductOrderLine OrderLine)`.

In order to determine the warehouse to ship from, this function would have to figure out warehouses that the product exists in, and then look at a number of associations in order to do its job (looking at the order associated with the product; the user associated with the order; then looking at the user's address).

An example of code for this may read like the example code shown below.

This is a "violation" of the principle. One of the problems here is that the method internalises a number of assumptions about how other bits of the system work (it knows about order lines, orders and users in addition to what it needs to know about).

Instead, if the method `Warehouse GetClosestWarehouseWithProduct(Product product, Address address)` it can do its work with many fewer assumptions and with cleaner code.

In this case, the first two lines are unnecessary.

Removing these lines mean that there are fewer potential problems which can occur within this code (for instance, if an Order does not have a User).

```
public Warehouse GetClosestWarehouseWithProduct(ProductOrderLine orderLine)
{
    Address orderShippingAddress = orderLine.Order.User.ShippingAddress;
    Product product = orderLine.Product;

    Warehouse[] warehousesWithStock = GetWarehousesWithStock(product);
    Warehouse closestWarehouse = null;
    decimal distanceKm = decimal.MaxValue;

    foreach (var warehouse in warehousesWithStock)
    {
        var warehouseDistanceKm = warehouse.DistanceFrom(orderShippingAddress);
        if (warehouseDistanceKm < distanceKm)
        {
            closestWarehouse = warehouse;
        }
    }

    return closestWarehouse;
}
```

MINIMISING COUPLING

The underlying coupling could be reduced even further by making it so that the method doesn't need to know about products at all (`Warehouse GetClosest(Warehouse[] warehouses, Address address)`). Now, the method becomes simpler and more algorithmic – it doesn't need to call an external method to find out which warehouses have stock, it merely finds which warehouse in a list is closest to a given address. At the same time, it becomes more general – it's now capable of finding out which warehouse is closest to any address.

Of course, here the consumer must retrieve the list of warehouses with stock and pass it into the method, but that's a reasonably clear thing to orchestrate.

BE EXPLICIT WITH FLOW

Code consists of both conditions and scenarios that are obvious to the reader, as well as those that are less clear. Consider the above GetClosestWarehouse method, which may look like the example shown below.

It is clear what happens when the "golden path" (the most frequent and expected case) is encountered.

It's a simple function, yet there are still a number of implicit flows that developers may not immediately see when looking at the method:

// What happens when there are no warehouses (in this case, null is returned).

// What happens when there's no address (in this case, the result is dependent on what the DistanceFrom function does).

// What happens when two warehouses are the same distance from the address (in this case, the warehouse which appears first in the list will be returned).

Each of the above cases is a good candidate for automated tests against a function like GetClosestWarehouse.

Each of the above cases is also a good candidate for additional changes to the code:

The resulting code is longer but tells the reader, upfront, what happens in each of the originally expected and anticipated cases. See the example on the next page.

```
public Warehouse GetClosestWarehouse(Warehouse[] warehouses, Address address)
        {
            Warehouse closestWarehouse = null;
            decimal distanceKm = decimal.MaxValue;

            foreach (var warehouse in warehouses)
            {
                var warehouseDistanceKm = warehouse.DistanceFrom(address);
                if (warehouseDistanceKm < distanceKm)
                {
                    closestWarehouse = warehouse;
                    disanceKm = warehouseDistanceKm;
                }
            }

            return closestWarehouse;
        }
```

```
/// <summary>
/// Finds the warehouse in the list that is the closest to an address
/// </summary>
/// <param name="warehouses">A list of warehouses to match against</param>
/// <param name="address">The address to calculate the distance to</param>
/// <returns>The closest warehouse, or null if no warehouses were provided</returns>
public Warehouse GetClosestWarehouse(Warehouse[] warehouses, Address address)
{
    if (address == null)
    {
        throw new ArgumentNullException(nameof(address));
    }

    if (warehouses == null || !warehouses.Any())
    {
        return null;
    }

    Warehouse closestWarehouse = null;
    decimal distanceKm = decimal.MaxValue;

    foreach (var warehouse in warehouses)
    {
        var warehouseDistanceKm = warehouse.DistanceFrom(address);

        // if two warehouses are the same distance, we keep the first
        if (warehouseDistanceKm < distanceKm)
        {
            closestWarehouse = warehouse;
            disanceKm = warehouseDistanceKm;
        }
    }

    return closestWarehouse;
}
```

COMMENT WHERE NECESSARY

One of the readability additions above was to add some comments (both XMLDoc and inline). The addition of these comments has aided the readability of the method.

TIP: Code quality's not just about the code that runs. Readability of comments is every bit as important as the code itself. Treat them in the same way – keep them up to date, keep them succinct, and keep them clear. They can and should point others to the rationale for implementation decisions, edge cases that were considered and assumptions that were made at development.

It's important to comment where necessary for clarity, but not to comment everywhere, as this reduces the signal to noise ratio significantly.

If the purpose of an operation is clear, there is little point in commenting it - "look at each provided warehouse" isn't a useful comment to put above the foreach statement.

One particular problem in a lot of applications we see is that comments quickly become outdated and are not maintained. Be sure to dedicate time to updating your comments when you update functionality.

It's frustrating and time wasting to read a description of how a function behaves that is contrary to the implementation.

If there are many places in a codebase where comments are out of date, engineers will stop trusting and reading the comments altogether, resulting in more time required to understand each piece of functionality, and hence slower overall development speed.

BE EXPLICIT WITH NAMING

The naming of fields, functions and classes is important, as these convey useful meaning before a developer starts to look at the related source code.

In the previous example, the DistanceFrom method is potentially poorly named, depending on what result we're actually looking for.

Its signature would be:

```
public decimal DistanceFrom(Address address)
```

This is not indicative of units, and is potentially ambiguous around meaning.

Even if the implementation was DistanceFromKm, this could be different to what we want.

An as-the-crow-flies distance may be reversible, but if the method had additional smarts that gave a road distance, the reversibility of the output is not guaranteed (one-way streets may result in significantly different distances depending on direction).

For this reason, DistanceToKm is a more descriptive and correct name. Alternatively, DistanceBetweenKm with a direction parameter would also be a good choice.

DESIGN BY CONTRACT, PRE-CONDITIONS AND POST-CONDITIONS

When developing functionality, care should be taken to consider the inputs, the outputs and any side effects that the code is expected to produce.

Each of these should be documented and make sense within the context of the application. This is its contract.

If a method lives up to its contract, it can be considered defect-free (in this case, issues are usually encountered when the naming or purpose of the functionality is ambiguous, or the consuming developer was unaware of the contract).

As such, if the contract is met, the implementation within a method is usually irrelevant for the purposes of doing anything outside of the method.

That said, a public method should not trust callers to provide it with valid data.

When data enters such a method, it should be checked for validity. In this way, the method can define what will happen in all cases.

Depending on how these results are structured (for instance if invalid data is provided, the method throws an exception), it is very clear to callers that they have done something unexpected.

Checking post-conditions and predictable side effects is something that is less frequent but serves as a useful litmus test.

Many languages provide some features that are useful for this checking, for instance C# has the Debug.Assert.

AVOIDING COMPLEX AND OFTEN-MISUNDERSTOOD LANGUAGE FEATURES

Every language has a number of areas where things are a little bit unclear, or where the language designers were unable to anticipate future development or changes.

One of the original C# designers, Eric Lippert, recently shared his 10 least favourite language features, which gives a decent starting point for some things to avoid in that language.

Some examples include:

// Empty statements – implicit no-op statements which do little except lead to unclear control flow and syntax errors. (for example while(true); contains an empty statement).

// Treating enums as integral types;

// Prefix and postfix increment and decrement operators. These should be avoided unless necessary or clear in usage (eg i++ in its own statement is clear enough);

// Finaliser/destructor logic – destructors are complex beasts that behave differently to what most developers would expect. When practically writing destructors, the code is often quite messy due to the lack of assumptions that you can make about the state of the object.

There are also a number of other complex/misunderstood features that are usually a good idea to steer away from:

// Using for statements where for-each will suffice.

// Multiple statement expressions in a for statement initialiser or iterator.

// Complex LINQ projections (sometimes).

Modern C# does not frequently need the for statement. By far the most common case in other languages is to look at each item in a list, in order. In this case, so long as the position of the object within the list is unimportant, a for-each statement is usually a better, and more clear choice as it introduces a named variable into the correct scope.

In the rare cases where for statements are necessary, it is almost never a good idea to place multiple statements in a for loop. Here is an example of valid, yet unclear C#:

```
for (int i = 1, j = 3; i < 4;i += --j == 0 ? 1 : 0, j = j
       == 0 ? 3 : j)
{
     Console.WriteLine("i:{0}, j:{1}", i, j);
}
```

Trying to guess what this code will do without stepping through the logic is difficult.

Moving the prefix increment operator (--j) out makes things marginally more clear, but doesn't help significantly.

The same statement can however be far more clearly written, avoiding the rare language features and much of the complexity as such:

```
for (int i = 1; i <= 3; i++)
{
    for (int j = 3; j >= 1; j--)
    {
         Console.WriteLine("i:{0}, j:{1}", i, j);
     }
}
```

The main point here is that virtually equivalent code can be written clearly, or unclearly, depending on how the developer chooses to go about using language features.

TIP: When readability changes are made, often the resulting code is longer. More code that is easier to understand and maintain is a trade-off that is almost always worth making. "Modern" C# is typically more readable than C# partially because it is more spread out.

CHOOSE THE MOST OBVIOUS APPROACH

Consider the following C# function:

```
public string SanitizePath(string path)

{

    return Path.GetInvalidPathChars().Aggregate(path,
   (current, invalidChar) => current Replace(invalidChar,
    '_'));

}
```

This function takes a string, obtains a list of characters that can't be used in a file path, replaces each of those characters in the given string with an underscore, and returns the updated string.

While achieving this in a single line of code may to some seem elegant or clever, those superlatives have come at the cost of clarity.

Most developers will gather the what of this method immediately, but many will not immediately understand the how of it, leading to a period of brow-furrowing and language documentation-delving.

The crime in this case is one of abusing a tool by using it for a non-intuitive purpose.

Aggregate() is a function that applies an accumulator function over a sequence. Its normal use case is a mathematical one; here we are taking advantage of its flexible nature to mutate a string.

Nothing about this usage of *Aggregate()* is intuitive, and because of this, some developers upon discovering this code are going to pause for longer than anyone should reasonably be expected to pause on a one-line function.

The time spent understanding this unnecessarily strange method does not come free.

Every piece of code written in this potentially confusing fashion is a piece of technical debt: it might have been quick and easy to write, but it costs time and money down the track.

Now consider the following functionally identical version of this code:

```
public string SanitizePath(string path)
{
    var result = path;
    var invalidPathChars = Path.GetInvalidPathChars();
    foreach (char invalidChar in invalidPathChars)
    {
        result = result.Replace(invalidChar , '_');
    }
    return result;
}
```

This version relies on common language constructs which will be familiar to even the novice C# developer, and as a result the code essentially documents itself upon casual inspection.

ENSURE METHODS ARE APPROPRIATELY SIZED

The ideal method length is something that is debated by software academics.

Some favour incredibly small methods (smaller than 15 lines) while others believe there's nothing wrong with longer methods of 100-200 lines.

Our pragmatic approach is that a method should be as short as practical, but not at the expense of functionality or clarity.

Each method should however only deal with one concept.

If an algorithm is necessarily complex and does not lend itself to being broken down, then separating out methods does not make sense.

On the contrary, if a method can be easily separated out into a number of private methods then this may make sense.

We however find that most readable methods tend to be somewhere between 10 and 100 lines.

Non line of business systems may however have different length tendencies. For instance, graphics routines often involve long and complex transformations which need to be performed in sequence – in this case, longer methods are not necessarily harmful.

TIP: There's no one-size-fits-all ideal method length. Keep your methods coherent, concise and related to a single process or concept, but don't break things down further unless there's some readability benefit. There is additional overhead in refactoring methods which have been split down past what is necessary.

MINIMISE SCOPE DEPTH AND EXIT EARLY

There are a near-infinite number of ways that any operation can be expressed.

Some of these are needlessly complex. Some of these are difficult to read.

One quick and easy measure to see how much complexity is hidden in a function is to look at the maximum number of scopes nested within each other.

Code that has more nesting is inherently more difficult to reason about.

Fortunately in many cases, it is straightforward to improve readability through slight modifications to the code.

We've seen a lot of legacy code like:

```
string BuildStatus(bool flag, bool flag2)
{
    string status = "No status";
    if (flag)
    {
        status = "Status";
        if (flag2)
        {
            status = "Status 2";
        }
    }
    return status;
}
```

Note that the underlying logic is far simpler than the code indicates.

If we minimise the scope depth and exit as soon as we know what we're returning, the method is much clearer.

See the example on the following page.

```
string BuildStatus(bool flag, bool flag2)
{
    if (flag && flag2)
    {
        return “Status 2”;
    }
    if (flag)
    {
        return “Status”;
    }
    return “No status”;
}
```

Reorganising code to have less nesting almost always aids in readability and hence quality.

IN SUMMARY

Code comprehensibility is an important part of code quality.

The faster people can read and understand your code, the more rapidly they can make effective and correct changes.

If your code actively works to point engineers at the edge cases and other design considerations that were incorporated during its development, this will help avoid the introduction of logic and design defects.

Making code readable can be done through a number of different techniques:

// Structural consistency comes from experience with design patterns, both Gang of Four and in-project structural patterns. It is important because it allows developers to make a lot of educated guesses about how functionality is likely to fit together.

// Stylistic consistency comes from having a well-defined and followed style guide, along with enforcement either in terms of automated tools or code reviews. If a codebase has inconsistent style, it naturally takes much longer to read and understand the code. Just as reading Elizabethan English is understandable to most English speakers, this understanding is slower because the text is unfamiliar)

// The minimisation of coupling, either by application of the Law of Demeter or by other means, helps keep things from breaking in unexpected circumstances and isolates parts of the system from changes in others. This means that refactoring doesn't touch on random and unrelated parts of the system. It also means when things break as a result of changes, they are more likely to be localised to one piece of functionality.

// Writing the code so non-golden-path flows are obvious allows others to change the code with more confidence. Comments are one such way to indicate assumptions, history or intent. Treating comments in the same way as code, in terms of readability, is a good idea but takes a lot of work.

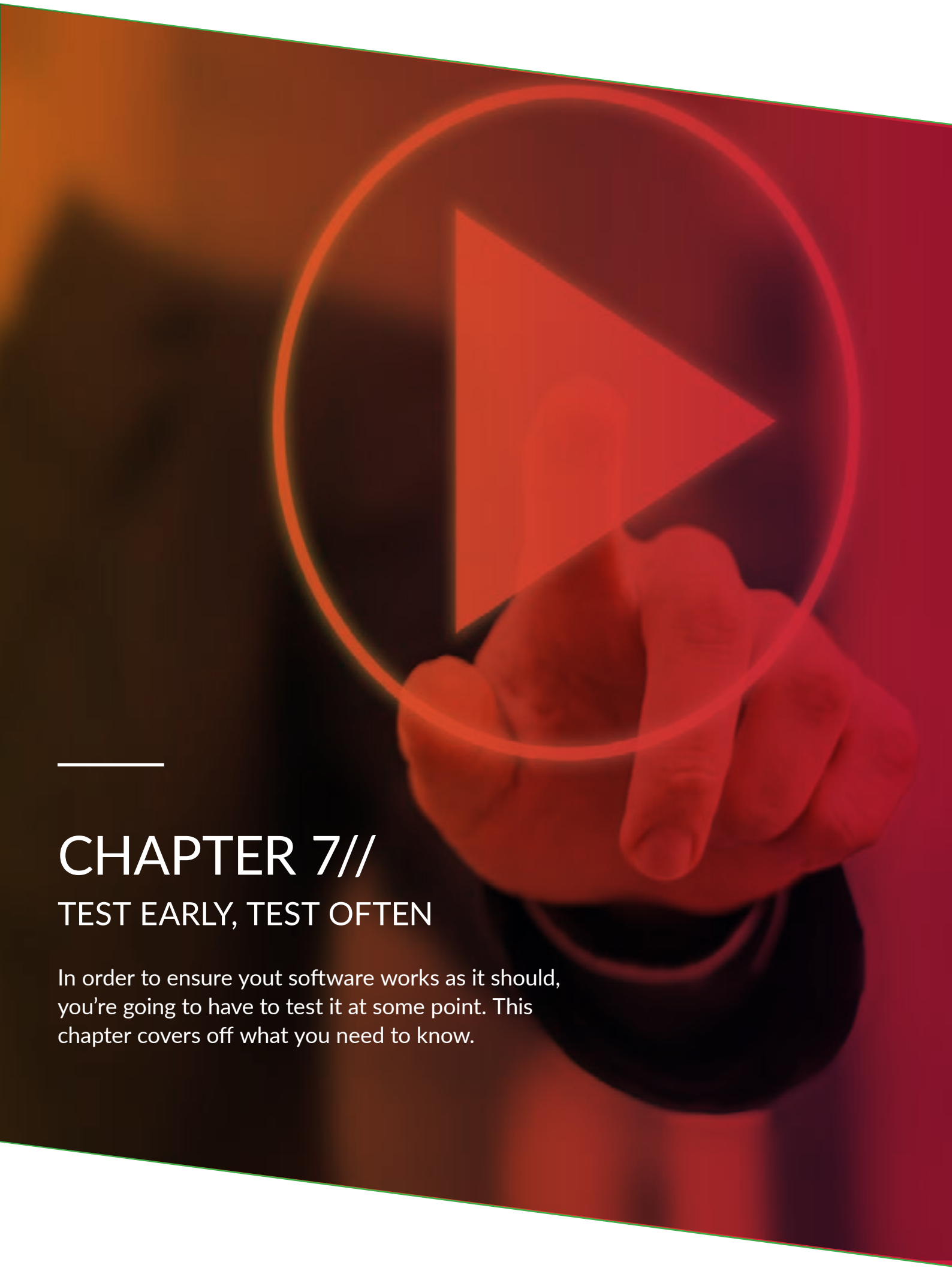
// Naming objects and classes is one of the more significant helpers or hindrances in understanding a codebase. Having poor names means that developers are more likely to make incorrect assumptions about the purpose or functionality of a method.

// When you treat methods and classes as contracts it aids consumers greatly. Inputs should be validated, defined side effects should be performed and outputs should be well defined. Language-specific assertions can assist in ensuring contracts are met. If a piece of an application clearly defines its contract, consumers can more easily trust that it works without needing to look at its implementation.

// Avoiding complex and often misunderstood language features results in codebases that are easier to read. Every language has a number of these, and in many cases linter tools, and code quality tools, will flag these with warnings.

// Methods should be appropriately sized. This however does not relate to a specific number of lines of code for all methods, as each has its own purpose. Methods should deal with a single concept and be as long as necessary to perform that task. If there are clearly defined sub-tasks, having methods for those may also make sense, however in many cases this is not possible. We would consider a method problematic if it were too big to be comprehensible as a whole.

// Keeping scope depth low usually results in more readable and easier to reason-about code. When a method needs to be rewritten or refactored to reduce complexity, try to reduce the nesting. What this often means is that when you calculate individual values, you'll want to separate them out into functions which can then return as soon as the return value can be determined.



CHAPTER 7//

TEST EARLY, TEST OFTEN

In order to ensure yout software works as it should, you're going to have to test it at some point. This chapter covers off what you need to know.

IN ORDER TO ENSURE THAT YOUR SOFTWARE WORKS AS IT SHOULD, YOU'RE GOING TO HAVE TO TEST IT AT SOME POINT. THERE ARE A MULTITUDE OF WAYS YOU CAN GO ABOUT DOING THIS TESTING; EACH WITH THEIR OWN ADVANTAGES AND DRAWBACKS.

We'll avoid an in-depth discussion of software development lifecycles, except to say that we've found an iterative/agile-like approach has worked best for our organisation.

One of the main advantages of an approach like this is we ensure when we develop functionality or make changes, these changes go through a number of different types of testing very quickly.

More traditional software development tends to defer the majority of (system) testing until a formalised feature-complete point.

At this point, the system is considered essentially complete, so the testers perform their tests, assemble a big list of defects, which the development team works to fix before handing a new version of the system back to the test team for another round.

This continues until there are no defects that will prevent the software from being shipped.

While this process should eventually result in software that has no major defects, it is far from an ideal situation:

// You don't really have any idea how much work is still required in order to be able to release the software. Resolution can take a significant amount of time, particularly if the defects are complex or systemic.

// Problems are discovered later on, making management-level planning more difficult and resulting in a higher chance of crunch-mode as deadlines loom.

// There is a non-zero cost to context-switching. It's much faster to keep making changes to something that you're currently working on than coming back to it days, weeks or months later to fix things up.

// There's less cohesion between project teams. Communication tends to be "over the wall" where one team hands everything over and considers their job done for the time being. Open channels of communication mean testers can ask questions and business analysts can be involved in coming up with solutions.

// The difficulties of changing direction once the system is essentially finalised leads to specification defects being treated as code defects. Often development and test teams are unable to decide whether an alternate solution meets customer needs. If it is not possible for analysts to weigh in on problems, non-ideal solutions to problems are often implemented.

// Having multiple phases which occur in sequence means the software takes longer than necessary, from inception to completion. In almost all cases, organisations would like their system to be built as quickly as possible, so this loss of efficiency is seen as a bad thing.

The two recurring problems are communication and predictability.

Testing earlier on in the process, and keeping the analysis team in the loop, reduces these issues significantly.

While an in-depth discussion on how to approach the many different types of testing sits outside the scope of this book, the approaches to unit and system testing have a large impact on the resultant code quality.

UNIT TEST

It always comes as a surprise to us how rare proper unit testing seems to be.

An effective unit test is one that places an individual piece of code (an algorithm, method or system) into a certain state, perform an action on it, and then ensure that the resulting state is as you expect.

Unit tests should be self-contained and isolate the component from others where possible. They should be fast to perform.

Writing repeatable and automatable unit tests is often seen, at least initially, as an activity that adds little value to the development process.

For this reason, it is often politically difficult to have time allocated to develop tests. This is doubly true for existing code.

TIP: If you can't find time for developing automated unit tests on existing code, try adding tests when changing code. Do this by refactoring code to be testable (i.e. change the structure and not the behaviour just yet!), adding tests to validate the functionality, and then making the changes.

When code is written alongside automated unit tests or written to be easily testable, it tends to be simpler, more readable and higher quality. The unfortunate side effect is that this testable code by its nature tends to look quite different to less-testable code.

Hands down, the biggest difficulty in introducing unit tests to existing code is that much code was not originally written in a way that lends itself to easy testing.

Codebases that were written 10 years ago are unlikely to use testing enablers such as dependency injection.

TIP: If adding unit tests to legacy code is problematic for you, we'd recommend Working Effectively with Legacy Code by Michael C Feathers – a book which focuses on making step-by-step changes to existing code in order to be able to introduce unit tests.

When adding test coverage, you reach a point where having all tests passing can give you confidence that the system will run as you think it should.

The benefit of having automated tests is in having tests fail when functionality is broken.

This can best be leveraged by running all automated tests during continuous integration builds or ensuring that tests pass before anything is handed to test.

Once the software engineers are satisfied a piece of functionality has been successfully implemented, software development should enter a system and feature testing phase.

SYSTEM AND FEATURE TESTING

Testers can verify the system largely works, while identifying areas that still need work.

During this time, we try to fix issues and get the fixes into the hands of the testers as quickly as possible.

We have found that keeping this process fluid and maintaining quick turnarounds on bugs results in both development and test teams rarely being idle.

It also saves us from expending unnecessary effort in logging and triaging different symptoms of the same defect.

Having test team involvement earlier in the process also ensures items that have been specified ambiguously are validated by another set of eyes early on.

Identifying these issues early means it is easier to bring business analysis into the conversation to resolve the ambiguity.

TIP: Treat system testing as an activity that will involve the BA, Development and Test teams. These teams working closely together will result in defects being noticed more quickly and being resolved correctly.

Even if it is not possible to have test involvement from the start of development, testing serves an important place in ensuring the quality of a system and codebase.

When a defect is raised by test, the underlying problem should be examined and fixed everywhere it occurs.

While this has a higher upfront cost, the ongoing costs will wind up being significantly lower.

A simple example may be a defect related to a file upload process which erroneously allows empty files to be uploaded.

Any fix to this defect should look at the other similar file uploads within the system to see if they also suffer from the same problem.

If possible, a fix should be implemented at the component level to fix not only all existing instances, which should then be tested themselves, but to also ensure the defect is not present on future file uploads.

In this case, the test team should add them to their test cases to ensure they check functionality for these types of files in the future.

It is through this process of testing early and having good dialogue between teams the quality of the solution and codebase becomes more important

TRACKING TESTING

Once the development of a feature is complete, it is important to track all found defects.

Note: there is a balance to be struck on the amount of detail that needs to be captured. There should be enough information to reproduce and troubleshoot the defect, however not so much information as to be burdensome to collect or create the bug in the defect log.

Fixes can, and should, be triaged and prioritised. This directs development efforts to the most critical problems first and means the test team is not blocked.

By doing this, more time is dedicated to actually testing functionality, which results in a higher quality product.

Over time, as the collection of defects grows, it is important to look at the list objectively to identify problem areas within the codebase itself.

A small component that has a disproportionate number of defects may indicate it is a prime candidate for additional tests, refactoring or other rework.

Having a large number of defects of one type, across many areas of the system, may indicate issues with approach, issues with staff training or the need for some additional code improvements.

CHECKLISTS

We find checklists to be helpful in situations where there's no good way to codify processes.

Checklists are a series of manual tests before e.g. code checkins; or a set of tests that testers should subject each piece of functionality to; or a list of smoke-tests to perform before making a deployment live.

These checklists can ensure consistency and quality across the software (both in the codebase and resulting system).

TIP: Try not to rely too heavily on checklists for day-to-day quality issues. They can serve well as a reminder for staff, but can easily be overlooked unless they are integrated into the system-building process.

IN SUMMARY

Testing is an important part of the software development process. The historically popular *waterfall method* resulted in testing processes which did little to ensure good and consistent code quality; Instead focussing solely on whether the resulting product was suitable for release.

Bringing testers and analysis together with developers throughout the development process means that defects can be identified and resolved more efficiently.

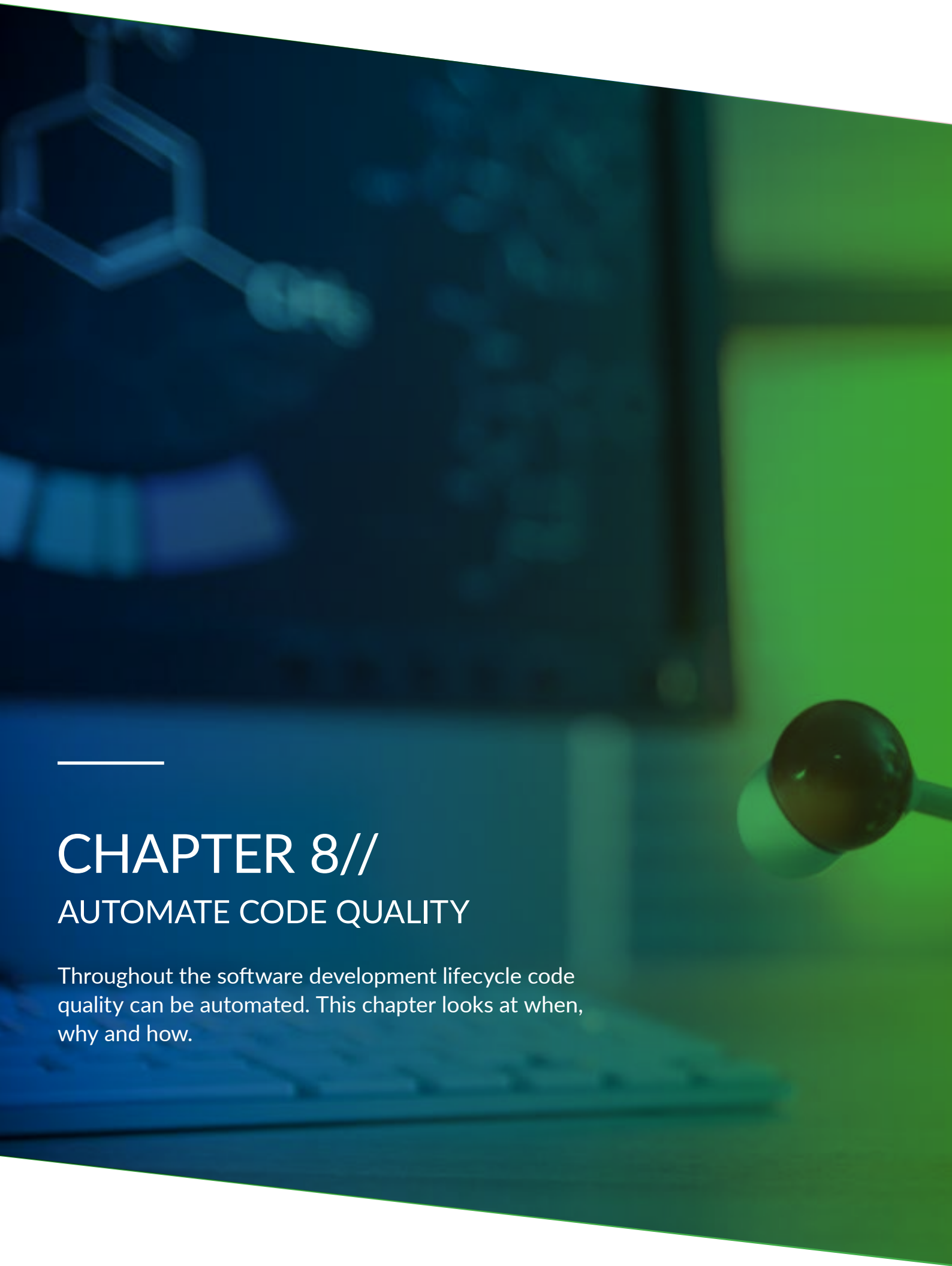
Developing automated unit tests not only ensures a bit of functionality works as expected, it also means if future changes affect the functionality of that part of the system, it will be clear and apparent earlier on in the process.

Once unit tests reach a certain saturation point, the team can have some level of confidence that the software will mostly work because all tests are passing.

Writing code in a way that can be tested by automated processes results in quite different code to code that is not written to be testable. Code that is written to be testable tends to be cleaner, simpler and better separated. This is another automatic win for the quality of the codebase.

Refactoring existing code so that it is more testable is an involved process. However, this usually brings significant readability and comprehension improvements. Building tests before modifying existing code is also a good way to ensure that functionality is not broken unintentionally.

The test process should result in a list of defects which can be analysed to identify trends. As such, defects found through test are something that can be used to feed back into the underlying code quality. A recurring problem indicates a different approach should be used. A series of unrelated problems in one component indicates the component should be separated out or simplified if possible. Where no automated solution exists to ensure a quality item, checklists can fill this gap.



CHAPTER 8//
AUTOMATE CODE QUALITY

Throughout the software development lifecycle code quality can be automated. This chapter looks at when, why and how.

THROUGHOUT THE SOFTWARE DEVELOPMENT LIFECYCLE CODE QUALITY CAN BE AUTOMATED. PREVIOUS SECTIONS HAVE TOUCHED ON SOME OF THE WAYS THAT THIS IS POSSIBLE. IT IS IMPORTANT, HOWEVER, TO KEEP IN MIND THAT EACH ACT OF AUTOMATION DOES COME AT A COST.

There are certain things that are too complex to automate, and things that are too difficult to properly automate. Where these occur, other strategies, such as code reviews and checklists, can assist.

Quality can be automated in a number of places:

// As code is being written

// When code is compiled or transpiled

// As standalone runnable tests

// At runtime (via checking preconditions and postconditions for methods)

// When code is committed

// During continuous integration builds

// Before code is turned into a release candidate

// During testing

While some tool types have significant overlap, they fall into the following broad categories:

// Linters

// Code style tools

// Static analysers

// Unit test frameworks and runners

// Assertion frameworks

// Checkin / commit policies and processes

// Automated testing frameworks and runners

// Runtime health monitoring and metrics

// Inspection platforms

LINTERS

Linters are typically used for interpreted languages. such as Javascript.

The purpose of linters is to prevent syntax errors and ambiguous or incorrect code from being written.

The tool will usually validate that variables have been declared correctly.

They will also pick up language syntax errors.

Many linters are opinionated and so will identify places where “strange” language constructs, such as Automatic Semicolon Insertion in JavaScript, are used.

Some take this further with opinionated style rules.

We have had success with JSHint, though ESLint does look interesting due to its pluggable architecture.

We’d recommend you incorporate linting on JavaScript resources within your software products.

These provide a first line of defence against the introduction of defects, either via merge problems or via unintentional changes by developers.

Linters are a good tool to get into any automated build process, for example as an action on a continuous integration build.

If dealing with an existing build, we would recommend excluding library files and relaxing as many of the rules as possible to get the number of warnings down to 0.

It may be practical to have two separate sets of rules for pre-existing files and new files, as even well-written projects are likely to have numerous warnings returned by tools like JSHint in its default configuration.

In addition to having linting happen on builds, we would recommend developers be familiar with linting prior to committing work. This way the feedback-fix loop is much smaller and faster.

CODE STYLE TOOLS

While both linters and code style tools are very similar and have crossover — both technically fall into the broader category of static analysis tools— they do have some differences.

Code style tools tend to work against compiled languages, and so are not interested in syntactical errors.

For C#, StyleCop is one such style tool. Although it is worth noting it is now partially deprecated due to the Roslyn Compiler/Service.

Jetbrains' ReSharper also contains style tools.

These tools, when configured, allow you to identify areas of code that are not in conformance with various code style guidelines.

While the tools have many rules, they tend to be somewhat inflexible and cannot necessarily encompass the complete contents of a style guide out of the box.

As such, developers should still be aware of the style guidelines within the organisation.

On greenfield projects, it may be simpler to conform to the default, or close-to-default, style provided by these tools.

While StyleCop started with an array of seemingly inconsistent styles, particularly when compared with eg code coming from Microsoft Developer Division, this was due to the history of the situation and is generally no longer the case.

There are still some rules we find good to turn off. For instance, we prefer to prefix our member variables and avoid using "this." throughout our code.

We find ReSharper matches our own guidelines well, and it also matches the majority of C#-based open source projects we have come across.

TIP: We've found it best to put our code style tools on developer machines. While we relax the style constraints within the linters and fail on lint errors, we tend to be softer on warnings coming from style tools as some of the fixes can harm readability in some cases. These decisions are something we'd recommend leaving up to the involved engineers.

STATIC ANALYSERS

Strictly speaking, static analysis is any analysis that is performed against the program without running it.

In practice, it is easier to exclude style and syntax tools from this definition as most other static analysis tools work against the compiled code.

For C#, FxCop is one such example of a static analyser. NDepend/SonarQube use static analysis but provide additional functionality as they are full inspection platforms.

FxCop is like StyleCop except it looks at design and architecture guidelines instead of code style. It also looks at the compiled assemblies rather than the source code itself.

It has been partially deprecated by the Roslyn compiler platform (Visual Studio has an inbuilt Code Analysis option that is effectively a new version of FxCop).

UNIT TEST FRAMEWORKS

There are many unit test frameworks out there, and any major language has dozens.

In C#, the main ones we've used are MSTest, NUnit and xUnit.net. For many tests the main difference between the three frameworks is in their syntax, and all three use a number of attributes to drive behaviour.

MSTest is arguably the product with the least functionality, followed by NUnit, with xUnit.net having (nice) support for things like filtered exception handling.

We like xUnit.net but do often wind up using MSTest on projects for various reasons, such as legacy concerns and the .NET 4.5 requirements for xUnit 2.

Similarly, there are some choices on how you go about isolating components, for instance how to come up with mock objects to test against). But these are largely a matter of personal preference.

Differences in frameworks aside, the biggest challenge with unit testing is to actually write the tests themselves.

It can be difficult to get approval to actually write tests; surprisingly few developers know how to write good tests.

Existing code usually needs to be changed to properly support testing, so it's often a bit of a rocky start. Even getting the test frameworks running in the build process is an important step.

All projects have some low hanging fruit that can be tested as-is.

When we were first looking at integrating automated tests several years ago, we started with some of our report processing logic.

We found that it was complex enough to justify tests to ensure we had correct results and it touched on an area that may have otherwise been problematic for us when we made changes.

In your codebases it may not be reporting, but there is likely a good candidate for where to start these tests.

Once you've tackled that, there are likely some other pain points which could benefit from having compile-time checks.

Soon enough, when combined with introduction of tests when new features are developed, test coverage will rise significantly.

ASSERTIONS

Assertions allow you to ensure preconditions, and to a lesser extent postconditions, are always satisfied.

It is helpful but not necessary for these preconditions to be simple – not-null or range-based conditions are straightforward for instance. Conditions libraries for C# include (Microsoft) Code Contracts, CuttingEdge.Conditions and Resharper Contract Annotations.

As of .NET 4, Code Contracts have been baked into the base framework. While the other two were historically good choices due to their lower barrier to entry, this is no longer the case.

Code Contracts allows you to express assumptions with code.

You can then perform automatic checking on the callers of these methods to ensure that the contracts are not violated at compile time.

Conditions can also be checked at runtime (though with one major caveat). Finally, as a beneficial side effect, Code Contracts can assist with XMLDoc generation.

Note: there is a set of significant drawbacks to Code Contracts (that precludes us from using them everywhere) – that the code must go through IL rewriting in order to support runtime checking.

This rewriting is not a trivial process and modifies the resulting code significantly. Similarly, there are performance concerns on rewriting.

Despite this, the static analysis provided by Code Contracts is very promising, and definitely an area that we will be working to improve our own capability in the coming months.

COMMIT POLICIES / POST-COMMIT CHECKS

Commit policies are less a tool on their own and more a hook point that allow you to validate something is the case prior to performing a code checkin/commit (or to notify that something was not the case immediately after a code checkin/commit).

This is a very convenient place to ensure:

// The code builds

// Interpreted languages lint

// The automated unit tests pass

// There are no build warnings

// That things are in a generally deployable state

In addition to this, we typically have daily integration builds which let us know that the code not only builds, but successfully deploys.

These environments can be used for non-development teams to verify behaviour and check progress, and are neither test nor UAT environments.

RUNTIME HEALTH MONITORING AND METRICS

It is important to know what your application is doing while it's running.

Tools that check uptime are handy to identify whether the application has critical problems, such as memory or power, which cause outages. But these are on the lower-end of 'useful', from a quality perspective.

Instead, the useful tools are those that measure application metrics, or those that 'kick in' when an application exception occurs.

ELMAH is an invaluable plugin for ASP.NET application error handling.

It provides visibility into errors that page visitors run into. We have successfully built out our own error handling platform from this.

Where possible, collection of usage metrics also brings in useful information about whether the software is fit for purpose.

It also informs where to best allocate resources or attention for future enhancement.

If lots of people are getting mid-way through a process and then dropping out on a particular step, that step may require some rework to improve the completion rate.

Also useful are periodic status checks for the various bits and pieces that the system relies on. As well as logging when these change.

These dependencies may be both internal and external. For instance, when an external provider deprecates and eventually turns off their API, you want to be aware of this.

Similarly, when an internal service is failing intermittently, it is important to know from a quality perspective. Both cases can be achieved by monitoring outside of the application itself, however these can each fail.

These may happen unexpectedly with little notice and may be an issue with the specific user account the application is using to connect to the service. Being notified something isn't working right now is not ideal. But is better than users running into the problem with you unaware.

INSPECTION PLATFORMS

There's not much to say about inspection platforms such as SonarQube.

They allow managers and developers ongoing visibility into the state of their application from a code quality perspective.

This is much the same way systems such as Visual Studio Team Foundation Services give a perspective of the time quality of the overall project. Other tools such as Atlassian Confluence and Kanban boards can help provide yet another perspective.

These tools can be used to gain an understanding at a high level of the amount of technical debt, and the amount of work that is likely to be required for a given release.

As discussed in the *Test Early, Test Often* chapter, it is important to try to get testing into the mix as early to ensure that the figures given by TFS or as a result in the planning boards are as accurate as possible.

SonarQube is interesting, because it provides a server-hosted place to see the status of code quality outputs at any given point in time, while also providing the ability to drill down to the project and file level.

NDepend is interesting in that it allows querying of compiled assemblies (and source code) via a query language called CQLinq.

These tools allow users to write queries to find e.g. unused public methods, methods which are long, and to perform other queries that help pinpoint areas that may be problematic.

TIP: CQLinq is a really powerful tool to query code. It allows you to use NDepend to gain insight into a codebase that would otherwise be impossible.

IN SUMMARY

There are a wide variety of types of tools that can fit into the development pipeline to help ensure the codebase and resulting product is of the highest quality. These tools range from the comparatively simple to complex standalone analysis services.

On the whole:

// Linting is useful wherever you have interpreted code to ensure that what gets used will actually run in practice. It is good to run this both at the developer machine and when code is committed.

// Style checkers are useful to ensure that code is consistent. This increases engineer comprehension and matches with internal style guidelines. Many of the things style checkers check for are subjective, so we typically just run these at the developer machine and keep things consistent with recurring code reviews.

// Static analysers (other than linters and style tools) look at the code or its output to ensure that things are structured well. These are useful tools to have on the developer machine, along with some relaxed settings on builds. They help keep code conforming to basic design guidelines.

// Unit testing frameworks and runners help ensure code doesn't break when things change. We recommend running tests as frequently as possible. Writing tests is difficult to justify at first due to its cost and list of perceived benefits. As time goes on, however, and as

the corpus of tests increases, their value goes up exponentially. Writing code with tests in mind results in cleaner and clearer code which is of a higher quality.

// Assertion frameworks and tools which check contracts hold a lot of promise. Our current opinion is that there is still work that needs to be done before incorporating a heavyweight contract system (Code Contracts) into our process. But it's currently one of the prime candidates for quality improvement within our own code.

// Checkin/commit policies and processes serve as hooks to run other processes. To us, running unit tests, linting and performing basic static analysis are essentials.

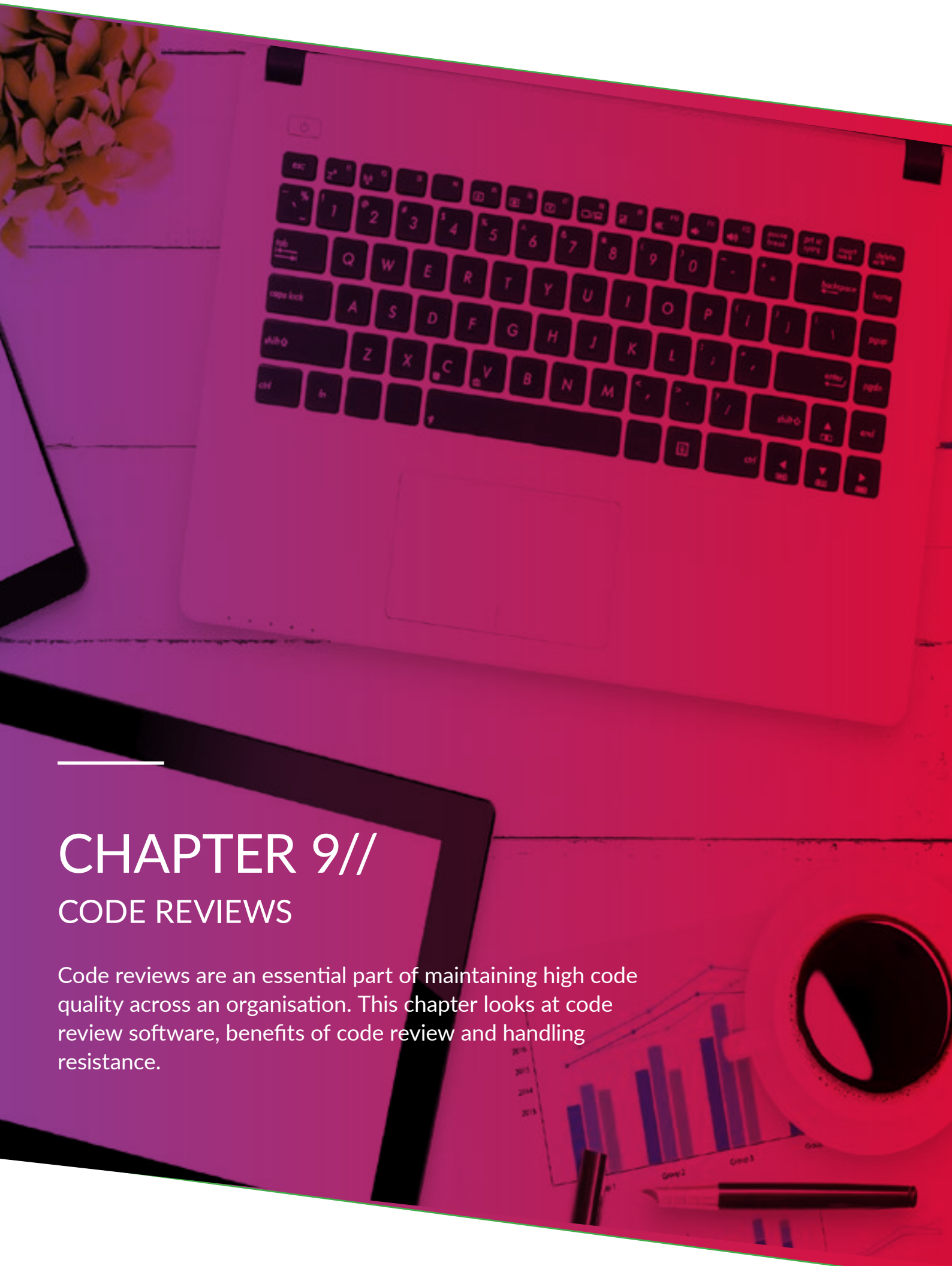
// Runtime health monitoring serves as a useful last-line-of-defence to let us know if something is misconfigured or not working. External monitoring can take you far, but there are certain things, such as account lockouts, that are difficult to anticipate/monitor for. Having monitoring built into the application itself ensures that you know when things aren't working.

// Inspection platforms provide useful management information that can be used to drive management and code focus related decisions. There are a number of options out there – but each has its own focus and gives its own visibility into the metrics of health in a codebase.

Remember tools are useful, and automating the use of tools is generally a timesaver.

With anything, however, there is always a trade-off.

It's important to ask how tools will help with any given process. Only spend the time and effort utilising tools that will help with the actual problems you need them to solve.



CHAPTER 9//

CODE REVIEWS

Code reviews are an essential part of maintaining high code quality across an organisation. This chapter looks at code review software, benefits of code review and handling resistance.

CODE REVIEWS ARE AN ESSENTIAL PART OF MAINTAINING HIGH CODE QUALITY ACROSS AN ORGANISATION. THEY SERVE A NUMBER OF PURPOSES – TO VERIFY THAT OTHERS ARE ACTUALLY WRITING CODE THAT IS READABLE AND MAINTAINABLE; TO SHARE IDEAS AND THOUGHTS ON MAINTAINABILITY; AND TO ENSURE THE QUALITY OF CRITICAL SECTIONS OF CODE.

Even with the use of automated tools, there are still big benefits to having another pair of eyes look over the code that is being written.

Sometimes developers have "tunnel vision" when it comes to problems, and solve the problem at hand but could have done something in a more generalised or straightforward way.

Sometimes there are things which read clearly to them but are incomprehensible to another engineer.

Sometimes there are places where an inexperienced developer is unsure about the best approach to solving a problem.

Sometimes there are bugs in newly written features that we'd like to catch before showing the system to the test team.

These are the cases where a code review is invaluable.

TIP: Build code reviews into your development process. Within F1 Solutions, we have weekly reviews involving every developer, and have found these to be an invaluable tool. The feedback that each developer receives must not be treated as criticism of their code, but taken as guidance on how to go about doing things from that point onward. Be sure also to provide a follow-up so that identified issues get resolved in the same way as other defects or quality issues.

CODE REVIEW SOFTWARE

There are a number of code review tools out there, including a code review task type and workflow built into Visual Studio Team Foundation Server.

While this process works well for some, the main goal of the review is to get people to look at code and give feedback. As long as the feedback is given and incorporated, we consider the review successful.

Over the years we've utilised a number of methods of reviewing code, including paper printouts with annotations, direct inline code annotation, assembly of a separate issues list, and the TFS workflow type.

Each developer tends to wind up having their own preference.

We have found this is something that doesn't have a big impact on the results of the review.

This is why we don't have any hard rules about how reviews are done.

We just ask there is feedback and that it is incorporated.

By mixing the reviews and performing them frequently, we have arrived at a place where everyone's review expectations are very similar.

BENEFITS

There are many benefits that stem from these regular code reviews.

The primary one is reviewed code is, by its nature, of higher quality than unreviewed code.

We find a lower incident of defects within code that has been reviewed. And after review, we find the code itself is more maintainable.

Often the changes that are proposed from a review are quite minor – clarifications, naming issues, or slight structural changes.

These are just some of the direct benefits of the review.

Sometimes the feedback consists of recommendations for action in the future – "X could have been done with Y".

Sometimes the review itself is for a new feature.

This is to give the reviewer some familiarity with it. This slowly improves the code of the reviewer and reviewee over time.

The flow on benefits from a review process are more profound. When reviews catch defects it means shorter dev and test cycles.

This means there is more room for other quality improvement processes, such as investing time into improving the build process or unit testing.

Having reviewers become familiar with parts of the system they did not write means they gain additional knowledge and will be familiar with the concepts and structure at a high level when they next need to work on that section.

Not only is a review a quality exercise, but it's also educational.

HANDLING RESISTANCE

Initially there was some resistance to widespread code reviews within our organisation.

Two main concerns were cited – interruption to work and bruised egos.

While it is true that reviewing code takes time, we believe the benefits brought by the review greatly outweigh their cost.

Occasionally we do have scheduling issues and there are problems getting all of the reviews done within our week time-frame, however with good project oversight this is a rare thing.

Similarly, we have found once the process got into full swing, egos were rarely an issue.

So long as the feedback being given is reasonable, people tend not to take things personally.

Of course, these two properties are dependent on the dynamics within any team.

TIP: Perhaps the best method to handle resistance to code reviews is to trial a couple. If the outcomes from the review are not generally useful, and the participants believe their time has been wasted, a different approach to these problems may be necessary. In our experience, we found once the reviews became frequent they have been effective and trouble-free.

ADVICE FOR REVIEWS

There are a variety of ways to conduct reviews, from formal meetings to more lightweight "pass code around" type review processes.

Formal meetings tend to have a bigger impact on schedules. They are also more difficult to organisation.

While it is more difficult to elicit a reasonable list of feedback from more lightweight reviews, it is possible to perform lightweight reviews more frequently.

This will usually result in more knowledge transfer; more bugs being resolved earlier and less interruption to work.

Giving the author the ability to select code is generally a good thing. It means they can pick an educational piece of code — they can find something they're not sure about. Or they can find something that is a suitable size — 100 to 300 lines of code is generally considered a good amount to look through in a review.

Note: some code is naturally faster to read, and may be more mundane, so applying rules is likely not necessary.

Checklists are invaluable in helping reviewers identify problem areas. They also encourage them to think analytically when developing code themselves.

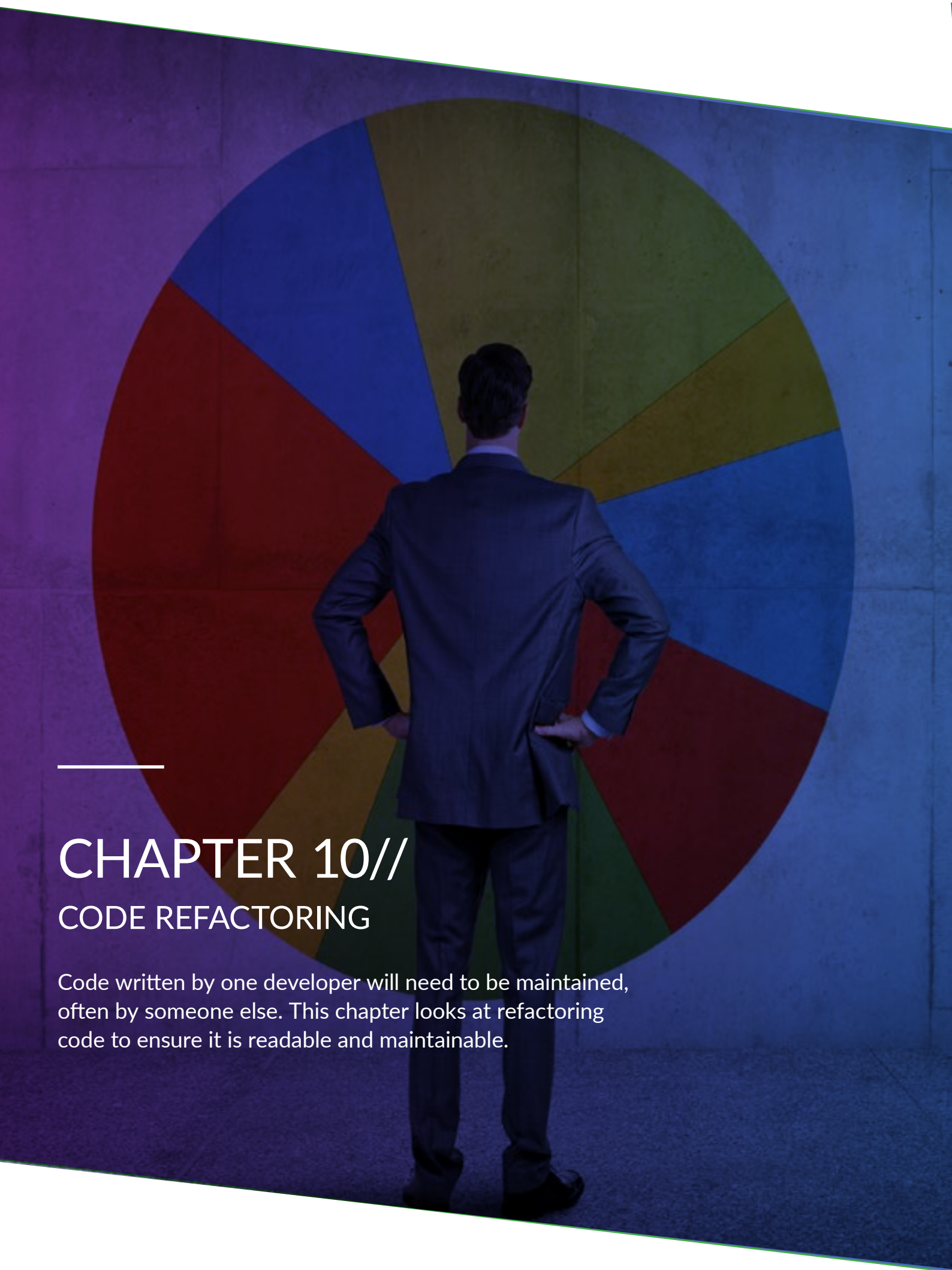
Having checklists also ensures after a few rounds of reviews, everyone's writing code that satisfies the items on the list.

IN SUMMARY

Code reviews are a valuable tool in more than one way. They provide a number of benefits immediately. They provide perspective, identify defects and find problem areas early on. Their benefit is also ongoing. This results in higher quality code over time, education of other developers and keeping levels of communication high.

There are definitely a number of ways reviews can be performed. However, it's often best to do things in as light and quick a way as possible to see if it's something that will work within a given organisation or team.

There may be some resistance to regular code reviews at first. However, over time, if it is quick and relatively painless, all parties should eventually realise it's a beneficial process. If code reviews did not work for us, we would likely look into other techniques that achieve similar outcomes, such as partial pair programming.



CHAPTER 10//

CODE REFACTORING

Code written by one developer will need to be maintained, often by someone else. This chapter looks at refactoring code to ensure it is readable and maintainable.

AN INESCAPABLE FACT OF SOFTWARE DEVELOPMENT IS CODE WRITTEN BY ONE DEVELOPER WILL NEED TO BE MAINTAINED, OFTEN BY SOMEONE ELSE. AS SUCH, ALL CODE SHOULD BE WRITTEN IN A WAY THAT IS GENERALLY READABLE AND MAINTAINABLE.

As was touched on in previous sections, most code will be read more than ten times as often as it is modified.

Due to business realities, it is, however, not always feasible to write ideal code all of the time.

These external pressures may come in the form of a missed or late-breaking requirement, or as a result of developers losing time to troubleshooting unexpected production issues.

Perhaps an engineer has been sick for a week, putting the team behind schedule.

Whatever the reason: there is pressure to get code written in far less time than originally anticipated.

To meet deadlines in these situations, it is tempting to cut some corners in order to get the required code shipped.

Some design guidelines or style rules may be skipped, or maybe methods aren't structured correctly.

With experience it is possible to identify the more "risky" shortcuts to avoid. But regardless it is essential developers find time later to come back and finish things.

This will save time and effort further down the track when the code next needs to be understood so it can be updated or extended.

Regardless of the cause, it is a simple reality that developers will be confronted with low-quality (or lower-quality) code from time to time; Either their own code or someone else's.

These are times when refactoring becomes essential.

First we will look at what is refactoring. Then we consider when and what you should refactor.

Lastly we look at some of tools to assist with refactoring.

WHAT IS REFACTORING

Refactoring is a code improvement process.

The goal is to change the structure of code in order to improve the maintainability of the code, rather than improving the performance or functionality.

This is achieved by decoupling modules, splitting big ugly methods into easy to read smaller ones and introducing new data structures or design patterns in order to simplify extension. In short, it is the process of restructuring to pay off technical debt incurred earlier in development.

The key feature of refactoring that differentiates it from other code improvement tasks is that functionality change is not the goal.

When you sit down to optimise code, you're looking to make the system more performant.

When you sit down to fix defects, you're looking to make the system function more correctly.

When you sit down to refactor, you're looking to make the system easier to extend and maintain in the future.

Like with many code quality related tasks, one of the major challenges with refactoring is it can be hard to quantify the end result.

Optimisation can be measured in improved performance; bug fixing can be measured by reduced bug count.

The results of refactoring are less tangible. It's an ongoing improvement in productivity in the future.

While tools can generate metrics that try to put metrics against code quality, such as code complexity figures or dependency graphs, it can be difficult to directly measure its impact.

TIP: Don't expect immediate results from refactoring. Unless you're refactoring immediately before performing functional changes, you're unlikely to see the efficiency benefits today or tomorrow. This is what makes dedicating time to refactoring difficult to justify.

WHEN TO REFACTOR

Even though refactoring is one of those tasks where the benefits can be hard to quantify, it is obvious to many developers refactoring is important.

Unfortunately, it can be difficult to convince managers or clients to spend precious project time and money on tasks that don't achieve a direct goal.

As with writing unit tests, performing refactoring as system changes are performed is an effective technique for improving a codebase.

For this, when an engineer starts work on a new feature, they identify areas of the code related to the new work which could benefit from additional clarity and simplification.

They then make structural improvements to the system and surrounding functionality by refactoring. Substantial improvements to a pre-existing system can be made without much budget impact in this case.

Further, the investment of time will pay itself off well into the future.

Refactoring can be risky.

One of the stated goals of refactoring is to not change existing functionality, and it is important that this holds true.

Automated unit testing is an effective technique to ensure that the underlying behaviour of the system remains unchanged.

Fundamentally, Test Driven Development is the marriage of these two ideas.

Tests should be written for each piece of functionality, as it is written or before it is changed. Once this is done, the code can be freely refactored and tidied, as long as all tests continue to pass. If that's the case, there is some confidence the functionality remains intact.

If, however, unit tests are not a possibility for a given project, there is additional risk in refactoring.

As such, in these cases it is always better to try to push refactoring toward the front of a development cycle.

This gives more time for the development and test teams to identify and resolve the problem. Starting a new feature by refactoring existing related code is a good way to go about things.

Conversely, heavy refactoring late in a development cycle is a risky proposition that should be avoided.

It increases the likelihood of issues being introduced. Also, the reduced test time means there is more risk of problems being missed and slipping into production.

TIP: Try to do major refactoring as early as possible during development, as refactoring is often a risky process. Avoid refactoring if possible toward the end as this can bear a high risk of introducing defects in to production.

WHAT SHOULD YOU REFACTOR

In order to increase the return on time spent refactoring, it is important to identify which areas of your code base will benefit most from being refactored. There are a number of factors that come into play.

// Age of the code.

// How often does the code change?

// Are there lots of bugs in the code?

Look at finding areas of your code where you regularly need to make changes.

A little refactoring to some regularly changed code can make for a marked increase in efficiency of development later.

Say you spend 4 hours improving a piece of regularly edited code, and as a result of refactoring you now only need spend 5 minutes making a change there instead of 30.

That time saving will quickly add up if you're making a lot of changes there.

Alternatively, is there a part of your code base people dread having to make changes to? Some old bit of data access, or clunky audit code that needs to be changed in 15 places any time a new column is added to the database?

These are areas that are prime for refactoring. Anything that requires making lots of complicated changes in order to do something small is prone to someone missing one of the steps.

If you can reduce those steps, you will make the odds of introducing bugs here much smaller.

Areas of the system that have been solid for a long time and are unlikely to be changed should be at the bottom of your list of things to refactor.

It can be tempting to go back and clean up some ancient code.

But if it's been sitting there doing its thing for the last 5 years without causing any problems, then you're likely to cause more bad than good by trying to improve things.

In this situation, the risk of introducing new bugs will often exceed the benefits to be gained from the improved code.

Code analysis tools can help with identifying parts of the system to refactor.

Using something like NDepend or Visual Studio to generate metrics on the maintainability of your code. Visual Studio's Code Metric Analysis will provide you with a per-project set of metrics, including cyclomatic complexity, class coupling, and lines of code, along with a general calculated "Maintainability Index."

These sorts of metrics can be useful for identifying areas within your solution that might be in desperate need of some refactoring love.

TOOL ASSISTED REFACTORING

Most IDEs have some built in functionality to assist with refactoring your code.

A simple example is renaming a method to something more meaningful. In addition to renaming the method, you will need to change every call to the method to use the new name.

Any decent text editor should have a basic Find and Replace function, which would help with this renaming task, but Find and Replace isn't very smart. If you told it to replace every instance of GetFoo with GetBar, it would fix all of your method calls, but it would also break every call to GetFood, or GetFooById, etc.

You could work around this by tweaking your Find query to something like "GetFoo(" instead of "GetFoo", but there will potentially be issues here too.

Plus, it means needing to figure out the best approach every time. This is where dedicated refactoring tools come into play.

Many IDEs support refactoring operations, such as Rename, as well as many others. Visual Studio comes with a set of common refactoring operations, which can be extended/improved through the use of plugins such as JetBrains ReSharper.

Refactoring tools will parse your code, adhering to the syntax of the particular language.

This means by instructing the tool to rename the "GetFoo" method on the "FooHelper" class, only legitimate calls to the method being changed will be updated, without effecting any "GetFoo" methods on other classes.

Some tools will even provide the option to update instances of the changed name in comments and string literals that are detected as being potentially related. This is useful for keeping XmlDoc comments up to date.

Other useful functions provided by many refactoring tools are

// Extract Method – This allows you to select a block of code and have the tool pull it all out into a new method. The tool will generally detect which variables need to be provided as parameters, and an appropriate return type. There are limitations to what can be automated here though, and the tool will usually allow you to modify the assumptions it has made prior to actually making the change.

// Change method signature – This will allow you to alter the signature of a method (change return type, add/remove/reorder parameters, change parameter types), and then assist you in updating locations where the method is called.

// Extract field/property – This will let you select a constant value used on a line of code, and move it into a variable of some kind. Useful if you've got some magic numbers lying around that should be constants, or if you want to make a string configurable.

Keep in mind that while refactoring tools can save you a lot of time and hassle, they are not perfect. Good tools will usually try to prompt you if they detect problems with the refactor attempt, but even so it is worth manually having a look over what has changed. They may sometimes do things you don't expect.

REFACTORING AND PERFORMANCE

Optimisation may be a side effect of refactoring, and vice versa. However, optimisation is not a goal of pure refactoring.

Refactoring aims to improve maintainability of code, whereas optimisation aims to improve performance. As you may have already thought, these goals can conflict.

There may be times where refactored code will be *less* efficient from a memory or computation standpoint than the original code.

This is not always a bad thing, depending on the exact situation. If the hit to performance is low, but the benefits of having the code more maintainable are high, then this is often a worthy trade off to make.

This is not carte blanche to ignore performance when it comes to refactoring though. If the code gets a lot slower, then it probably isn't worth making the change.

Always be mindful of any potential performance issues that may be introduced by refactoring.

When breaking code up into smaller methods with single purposes, it may be harder to notice nested loops, which can easily cause computation time issues with large data sets.

When you have a doubly nested loop and add another loop inside, it's quite obvious that you now have 3 loops nested.

On the other hand, if you have a method being called inside a loop, and then later add a loop to that method, you might not realise straight away that you've gone from O(n) to O(n^2), or even worse, from O(n^2) to O(n^3).

EXAMPLE OF REFACTORING

To illustrate what refactoring could look like, below is a contrived example of some code to process a payment.

BEFORE

```
public void ProcessPayment(PaymentInfo paymentInfo)
{
  switch (paymentInfo.PaymentType)
  {
    case PaymentType.Cash:
      var cashPayment = new CashPayment();
      cashPayment.Amount = paymentInfo.Amount;
      DatabaseContext.Save(cashPayment);
      break;
    case PaymentType.Credit:
      var creditPayment = new CreditPayment();
      creditPayment.Amount = paymentInfo.Amount;
      using (var processor = CreditTransactionService.DefaultService)
      {
        var result = processor.ProcessOnlinePayment(paymentInfo.CardName,
                                                    paymentInfo.CardNumber,
                                                    paymentInfo.Amount);
        creditPayment.TransactionNumber = result.TransactionNumber;
      }
      DatabaseContext.Save(creditPayment);
      break;
  }
}
```

This example contains two separate logic paths in the one method. If we had to extend the logic for processing either of these types of payment, this method would start getting even longer and messier.

The below code is functionally the same as the above, but separates the logic into extra methods. This is an example of how refactoring code can make your code easier to extend later on.

LOGIC SEPERATED INTO EXTRA METHODS

```
public void ProcessPayment(PaymentInfo paymentInfo)
{
  switch (paymentInfo.PaymentType)
  {
    case PaymentType.Cash:
      ProcessCashPayment(paymentInfo);
      break;
    case PaymentType.Credit:
      ProcessCreditPayment(paymentInfo);
      break;
  }
}

private void ProcessCashPayment(PaymentInfo paymentInfo)
{
  var cashPayment = new CashPayment();
  cashPayment.Amount = paymentInfo.Amount;
  DatabaseContext.Save(cashPayment);
}

private void ProcessCreditPayment(PaymentInfo paymentInfo)
{
  var creditPayment = new CreditPayment();
  creditPayment.Amount = paymentInfo.Amount;
  creditPayment.TransactionNumber = PerformOnlineCreditProcessing(paymentInfo);
  DatabaseContext.Save(creditPayment);
}

private int PerformOnlineCreditProcessing(PaymentInfo paymentInfo)
{
  using (var processor = CreditTransactionService.DefaultService)
  {
    var result = processor.ProcessOnlinePayment(paymentInfo.CardName,
                                                paymentInfo.CardNumber,
                                                paymentInfo.Amount);
    return result.TransactionNumber;
  }
}
```

Here each bit of logic is encapsulated in their own methods.

If we needed to add more functionality to the online credit processing, we can easily make that method longer without making the other areas of code less readable.

This was a very simple and contrived example. It could even be extended further.

Processing logic could be split into dedicated processor classes loaded by dependency injection, based on the type of the payment.

Database save logic could be split out into its own area to allow for consolidation of validation logic (validation is missing from this example for brevity).

IN SUMMARY

Refactoring is an important part of software development.

While it doesn't make a noticeable change to the software from an end user perspective, it helps to reduce costs of future development.

Regular refactoring of your code will make your life, as a developer, easier.

Your clients will also benefit by saving them time and money in the long run.

You should do your best to find time for refactoring, whenever possible.

Ideally it should be early in a development cycle.

Focus on refactoring code prone to change, and which is currently difficult to maintain.

Ensure refactoring does not introduce new functional issues or harm performance of the system.

CHAPTER 11//

FINAL THOUGHTS

Managing code quality is a long term game.

THIS BOOK HAS APPROACHED THE TOPIC OF IMPROVING CODE QUALITY FROM A NUMBER OF DIRECTIONS. WE HAVE OUTLINED, AT A BROAD LEVEL, SOME OF THE CHALLENGES OUR ORGANISATION HAS FACED IN THE PAST AND HOW WE HAVE WORKED IN RESPONSE. THE RECURRING MESSAGE IS BEST PUT AS AN ANALYTICAL ONE — EVERY TEAM AND PROJECT IS DIFFERENT AND THERE ARE NO ONE-SIZE FITS ALL SOLUTIONS.

We can make no guarantees that the techniques that have worked for us will work in every team.

Indeed, that is almost the point: *there are a plethora of options.*

Many of them address a singular issue.

The best way to improve code quality in any team is to look at a list of the biggest quality problems and to figure out how best to address those individual problems.

We have some recurring elements within our development process, for example our development stack (slightly modified if required) works well for us.

It is designed for security and robustness, and is something that is appropriate for every scenario we have built our systems for.

We think that it's a solid foundation we're building on, based on our situation.

Similarly, our process for code analysis benefits us well.

We have rejigged our requirements elicitation process and testing processes to work well with our development team and to ensure maximum communication.

In short, we have tuned our process to deal with the problems we have historically faced. Any other team will, of course, be facing different problems.

If a team spends significant time reworking things that they've already developed, there may be a communication issue at play.

If a team is constantly committing broken interpreted code, introducing linting will help.

If development is inexplicably slow, it may be a tooling, codebase complexity, or comprehensibility issue.

Often the first steps toward improving code quality will be relatively obvious and reactionary in this way — a problem with an apparent solution.

As things progress, however, the opportunities for improvement of the codebase's quality may become less apparent.

In these cases, it's useful to collect and use metrics to objectively determine what needs attention.

In a lot of cases, the results of reducing technical debt and improving code quality will not be apparent immediately.

The true benefit is not immediate. But slowly builds up over time. Altering code for readability takes time.

But the 5 minutes each engineer saves when looking at that code in the future quickly begins to add up. Changes that may have taken days in a poor codebase can be done a couple of hours on a high quality one.

So the other side of the message is one of persistence.

Apart for some obvious low hanging fruit, improving code quality is something that takes work and time – do not be deterred if quality payoffs do not occur immediately.

Managing code quality is a long-term game.

We hope that the content of this book has been insightful and useful to readers.

It's through these processes that we have refined our own development technique, methodologies and processes for maximum efficiency.


F1 SOLUTIONS