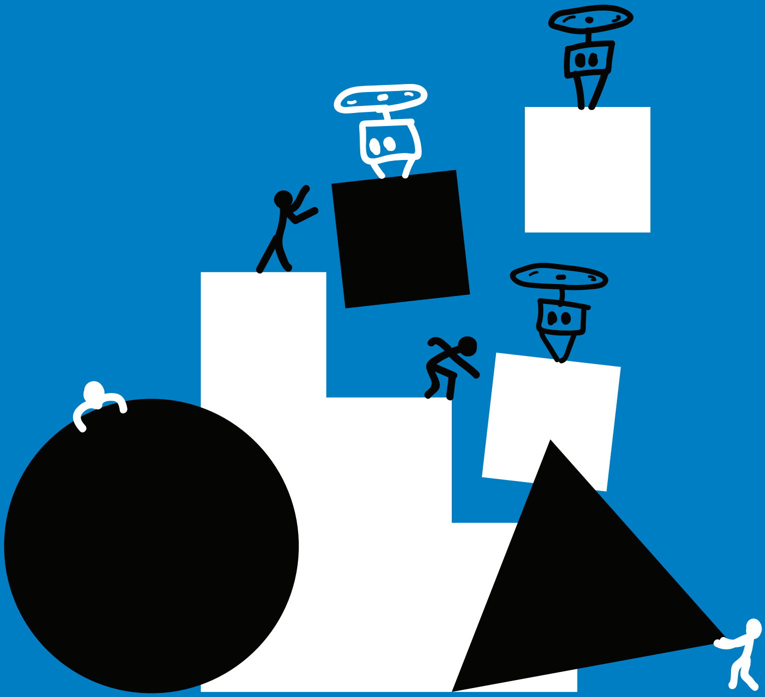


Bots that code.

The Continuous Modernisation Playbook

Eban Escott   Indi Tansey

CODEBOTS

Version

hydrogen

A big thanks to all the contributors for this book including Isaac Joe Kong, Naveen Kumar, Tessa Holland, and Kellie Lutze. Also a big thanks to Paul Strooper, Gary Buck, Dave Duncanson, Jon Yeo, and Trent Buck for their reviews and input into shaping this book. We love having you all in our community!

Codebots Pty Ltd
Office 1, 55 Railway Terrace
Milton, QLD, Australia, 4064
`https://codebots.com/`
Please contact distribution at `mailto:support@codebots.com`
ISBN 978-0-646-80083-7

Contents

Preface **5**

1 Introduction **11**

1.1 Hypothesis Statement 18
1.2 Business Agility 18
1.3 Software/People Fit 21
1.4 The Insanity of Legacy Systems 24
1.5 Knowledge is Power 25
1.6 Bots that Code 29
1.7 Future of Work 31
1.8 Summary 32

2 Legacy Systems **35**

2.1 Root Causes 37
2.2 Inescapable Truths 40
2.3 Firecracker migration pattern 45
2.4 Divide-and-Conquer Migration Pattern 47
2.5 Summary 51

3 Strategy and Tactics **53**

3.1 Science + Iterations 59
3.2 Community 63
3.3 Microservices 67

3.4 Fog of War 71
3.5 Modelling 74
3.6 Summary 81

4 Codebots 83

4.1 State and Behaviour 90
4.2 Entity Diagram 92
4.3 User Interface (UI) Diagram 94
4.4 Security Diagram 96
4.5 Development Target 97
4.6 Testing Target 106
4.7 Summary 112

5 Migration Kit 115

5.1 How to Use this Kit 116
5.2 Reverse Engineering Requirements 118
5.3 Experimental Framework 123
5.4 Community Onion 127
5.5 Managing Expectations 132
5.6 Story Estimation 138
5.7 Entity and Requirements Traceability Matrix 144
5.8 Database Migration 150
5.9 Spreadsheet Migration 153
5.10 PDF Migration 158
5.11 Bubble Context and Anticorruption Layer 166

6 Conclusion 173

Glossary 181

Acronyms 191

Bibliography 193

Appendix A Change Log 199

Appendix B Codebots Field Trials 201

Appendix C Behaviours 205

Appendix D Agile Manifesto 209

Appendix E Risk Register 211

#1 Inaccurate Estimates 212
#2 Scope Variations 213
#3 End-user Engagement 214
#4 Stakeholder Expectations 215
#5 Poor Quality 216
#6 Poor Productivity 217
#7 Inadequate Risk Management 218
#8 Low Partner Engagement 219
#9 Inadequate Human Resources 220
#10 Lack of Ownership 221

Preface

“You will find only what you bring in.” Yoda

We embrace the continuous modernisation mindset in everything we do, from building software to the publishing of this book. To avoid this book becoming outdated, we modernise it as market conditions change, sometimes multiple times per year and we will never consider it will be complete. Through the use of experimentation, and feeding the successes and failures of real world projects back into the book, it is our goal to make this the most relevant and well-read book on transforming legacy systems across the planet. Each time a new edition is published, we will send out the latest version to everyone who has purchased the book previously. [1] This is our commitment to continually keep our community up to date as the technology and tactics of this book evolve.

The tone of this book is a reflection of our four company values illustrated in Figure 1. As leaders, we are passionate about core values and recognise that values drive culture, culture drives behaviour and behaviour drives the performance of our entire community.

[1]Go to `https://codebots.com/continuous-modernisation` to make sure you have the latest version or contact us at *support@codebots.com*.

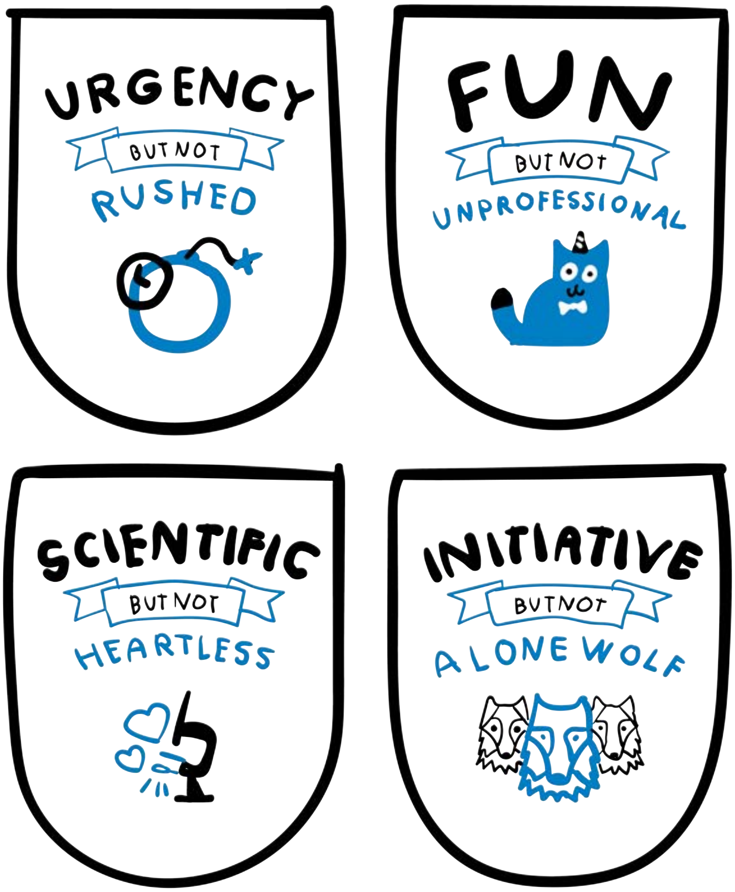


Figure 1: Our Values

The intended audience of this book is medium to large enterprises that have many software systems including off-the-shelf and custom software applications. If you are from a small business and you are struggling with your software applications, then you may find this book useful too, but small businesses usually do not have the budget required to move away from cheaper off-the-shelf applications and spreadsheets. Our intended audience is:

- CEOs and business managers that want a broader understanding and strategy the role software systems can play in the organisation.
- CTOs and IT managers that are involved in making decisions around the organisational direction and use of software systems.
- Any other stakeholders (designers, software engineers, business analysts, project managers, etc) that are involved in software projects and want to do it better.

In our approach to the book, you will see that we build on familiar technology concepts and business methodologies but take the initiative to repurpose them for the goals of continuous modernisation. We act urgently, with a strong bias to action and we prefer to test our raw assumptions and get feedback, rather than crafting something that no one needs or wants to read. We develop our opinions by delving into the facts, balancing science and data with human empathy to find insights that help us make better decisions. Finally, we are hard-working professionals striving to achieve our mission but we never forget to have fun along the way :)

In this book you will also find a mix of both technical information about software and also cultural stories around people. Companies have become reliant on an army of people who know their legacy systems so the challenge for building successful projects is as much, if not more, a people story rather than a technical one. This explains why our intended audience for the book is decision and strategy makers who want to wrestle back control, gain clarity and learn tactics to drive successful digital transformation projects. We are eager to learn how our readers use the migration kit activities presented in a later chapter and plan to share specific examples of how the kit has been applied in future editions.

We love books and inspirational characters like Patrick Lencioni who creates The Advantage through organisational health [21], Simon Sinek and his book, Start With Why, about how great leaders inspire action [30]. Plus, business bibles like Scaling Up from Verne Harnish [15] that cover the full gamut of people, strategy, execution, and cash for creating successful companies to name just a short few.

The heroes' journey is a universal story, that involves a hero who goes on an adventure, and in a decisive crisis wins a victory, and comes back transformed. The Continuous Modernisation story starts here, with you, so pour yourself a cup of tea, open your mind and join us in applying the theory and tactics that will advance your business agility journey.

1. Introduction

“Knowledge is power. Information is liberating. Education is the premise of progress, in every society, in every family.” Kofi Annan

Technology, once the domain of a team under the CFO has now become its own thing alongside the CTO and CIO. Technology is driving change at every level and every part of an organisation. But there's a challenge. We have created a huge amount of immovable software systems that are now inhibiting further change. Across the globe today, legacy systems are one of the biggest problems faced by the software industry. The problem cannot be overstated, and as we travel further into the future, legacy systems could arguably become the most common problem across all business sectors and industries.

As a leader, how do you keep abreast of the innovations and rapid changes in your market? Furthermore, how do you ensure that your software systems will allow you to meet changing market conditions? The days of relying on technologists without any business insight are over. If you know that technology now touches every part of the business, a strategy is needed that takes into account the impact it has on the business from a productivity, culture, and agility point of view.

Most organisations have reacted with a program of work around a digital transformation but many of these programs are failing. These types of projects carry a lot of risk as there are many moving parts to them and they generally touch multiple parts of the business at the same time. To mitigate the risks you must deeply understand the root causes of the problems else you will be leaving it up to chance for a successful outcome.

Continuous modernisation is a set of strategies and tactics that can help an organisation navigate the pitfalls of digital transformation projects. One of the tactics includes the use of bots that code. Codebots are software robots that write code alongside your human team. On average, they write over 90% of the code and teams using a codebot are 8.3 times faster. Some of the benefits include:

- Quality at speed.
- Reuse at scale.
- Reduce costs.
- Automated migration.
- No vendor lock in. You own the source code.
- Proven case studies.

Before we dive in too deep, let's set the stage and do a thought experiment on how do we build software applications today. In this thought experiment, we will play the role of the software team and you will be the product owner. We have assembled a cross functional team with a mix of skills including design, organisation, and technical. You are an expert in your domain and you have built up years of experience understanding the good and the bad of your industry. Overtime, you have suspected that there is a better way and you are finally ready to build a software application to bring your ideas to life. Exciting times!

We begin the process with some healthy and robust conversations. These conversations would happen in various locations like in a meeting or possibly over the phone. We would no doubt be sending emails backwards and forwards to each other as we scope out and refine the ideas. We would also do some long whiteboard sessions and draw some meaningful diagrams to represent the ideas and the solution. All of this would be put into some sort of requirements specification that represents the plan of what we are going to build, test, and launch. The software team would then take the specification and translate it into code.

As seen in Figure 1.1, we have created a body of knowledge about the software application but the connection is loose between the knowledge and the code. It is the responsibility of the developers to translate the requirements into the code. In practice, the specifications are usually stored in the bottom draw of the developers desk and largely ignored. Organisations that are embracing Agile are helping with this problem by involving the team throughout the entire lifecycle but the connection between the body of knowledge and the code remains loose.

The loose connection creates a lot of risk and makes the individual developers a bottleneck for the entire process. There is a significant amount of knowledge being lost and a lot of manual labour occurring that is ripe for automation. With each individual software application being treated as a snowflake, we are setting ourselves up to make the same mistakes over and over.

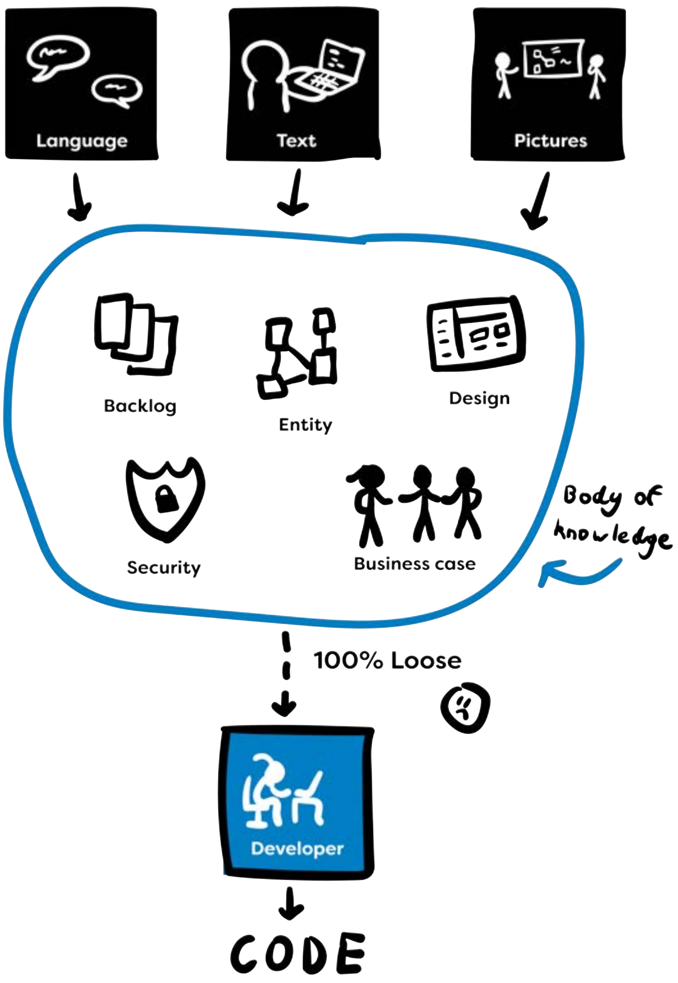


Figure 1.1: Typical software development project

Now let's imagine an alternative scenario. Let's take that requirements specification out of the bottom draw of the desk and put it back on the table in front of us. That body of knowledge is extremely important on many levels. We have had many conversations using language and captured a lot in text. We have drawn a lot of meaningful diagrams that visually represent our intent. Now here is the big difference, imagine a world where it is not just yourself and us that can understand the knowledge, but a world where a codebot can understand it as well. The codebot can then take that knowledge and write the vast majority of the code required for the solution.

As seen in **??**, this creates a strong connection between the body of knowledge and the code. The codebot is able to write the vast majority of the solution and this opens up all sorts of possibilities. Software applications can be made faster, of better quality, and we can maximise reuse across multiple applications. But it also leads to a fundamental shift around the role of knowledge and the people within the organisation.

By making the connection strong between the knowledge and the code, we are declaring that the knowledge is as important as the code itself. This premise has multiple and far reaching consequences. It can be the foundation for constructing a new way of looking at some old problems. With an open-mind, let's put forward a hypothesis and use an experimental framework to see what new boundaries we can discover.

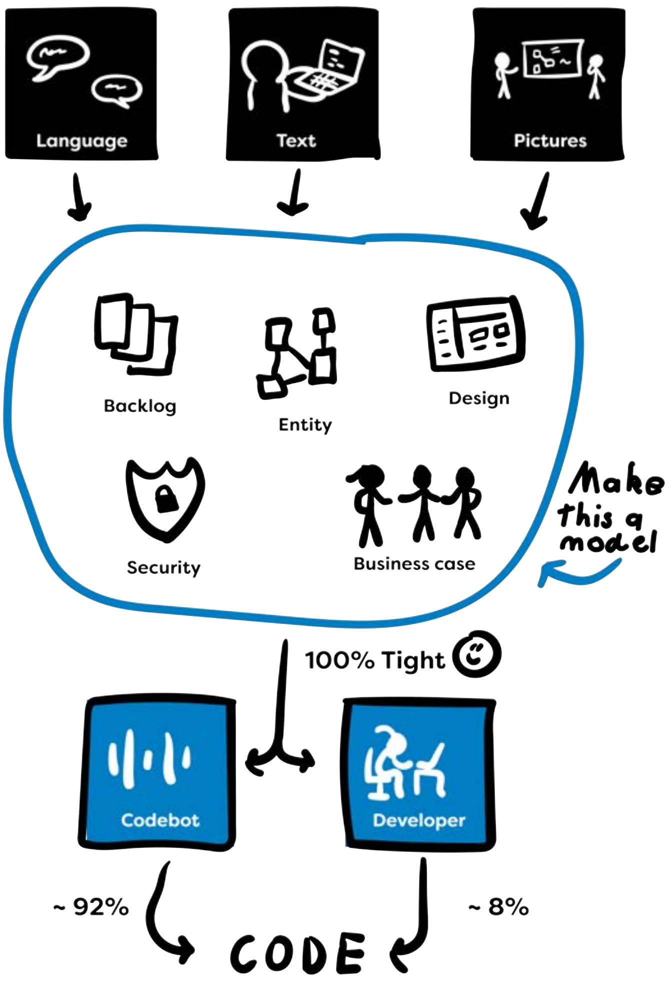


Figure 1.2: Software development project with a codebot

1.1. Hypothesis Statement

The hypothesis presented in this book is that business agility can be positively impacted through a process of continuous modernisation. Continuous modernisation is a strategy for legacy systems to ensure the *software* fits the *people* using it. The problems faced include technical challenges around the *software* and cultural challenges around the *people*. When a software/people fit is found, momentum is gained by the organisation and business agility is increased.

1.2. Business Agility

A digital transformation implies a one-off event, but in practice, a digital transformation should just be the beginning. We prefer the term continuous modernisation, and as the name implies it is a mode of evolution that does not stop, it's an ongoing process.

As an organisation scales up, its complexity grows. An example of this occurring is the communication lines between people. When there are 3 people in the organisation there are 3 communication lines. When there are 4 people there are 6 lines, when there 5 people there are 10 lines, and when you start hitting some decent size like 50 people, the communication lines are 1225. It is humanly impossible to keep track of this many communication lines and they grow exponentially with more people. This complexity generates three fundamental barriers around leadership, scalable infrastructure, and marketing [15].

These fundamental barriers create valleys of death that occur at certain organisational sizes and makes it is very difficult to grow and stay relevant. If they are not addressed, the overall result is a loss in momentum and business agility. Simply put, the organisation cannot manoeuvre to meet changing market conditions and is likely to find it harder and harder to deliver value on business outcomes.

Legacy systems are a main contributor to a loss in business agility. A typical example (and one we all know too well) that many organisations face is around reporting. When there are a number of legacy systems around different business units, there is usually a lag time in getting the appropriate data collected and subsequent reports to management. Sometimes the lag time can be weeks if not months depending on the size of the organisation. This has a negative effect on the organisation as the leaders cannot get access to the right information at the right time, hence decreasing business agility.

Another of the more frustrating scenarios is around innovation. Imagine that someone in the organisation has a great idea that will clearly make a difference as they have run experiments and collected data to prove their business case. However, the software that underpins that section of the business is a legacy system like something off-the-shelf or an old custom system that simply cannot be updated to match the idea. The organisation loses out on many levels here. Firstly, the idea is lost and the organisation does not adapt to a changed market condition. Secondly, the actual person who had the idea is deflated and this has a negative impact on the culture. Especially if it happens often.

These are two examples of how legacy software can have a negative effect on the organisation. And I am sure that you can think of a few more. Overall, it is clear to an experienced practitioner that legacy systems have an overall negative effect on your organisation's business agility. So, what is the solution to such a complex problem? Or even, is there one? In reality, there is no silver bullet, but with a greater understanding and education around the main contributors, it is possible to use a playbook of strategies and tactics to minimise the impact of legacy software on business agility.

To address the problem, organisations must first acknowledge and really think about how they can increase business agility. As seen in Figure 1.3, as the market conditions change, the legacy software can eventually plunge the organisation into a valley of death.

So, what does it mean to continuously modernise? Firstly, it recognises that the moment you start using new software in an organisation, it is a legacy system. And as time progresses, the longer you leave the legacy system without giving it the love and attention it needs, the harder it will be to update it and the more business agility will drop as a result, as illustrated in Figure 1.3

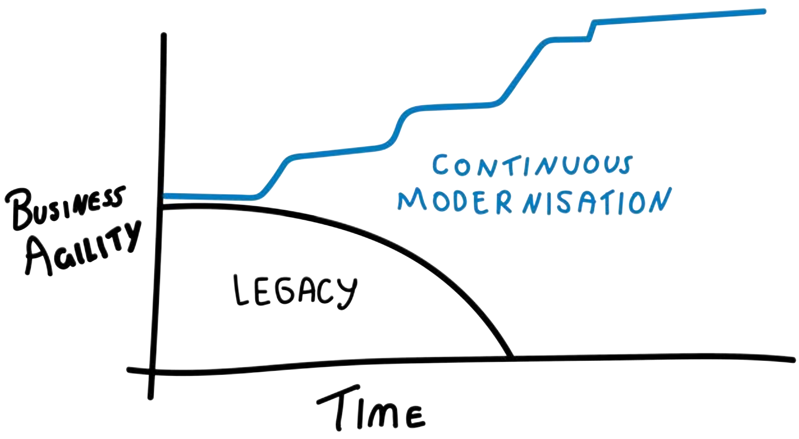


Figure 1.3: Continuous modernisation increases business agility over time

1.3. Software/People Fit

A lean enterprise [27] is always looking for a product/market fit. The build-measure-learn cycles allow startups to iterate from a Minimum Viable Product (MVP) to find traction and build a highly scalable business. Great idea! But how do organisations use this in context of an enterprise? A lean enterprise [16] is an extension into larger organisations to help them innovate at scale. By looking at the mechanics of how enterprises work, it is possible to influence the technical and cultural forces of an organisation to move them into a more lean and agile mindset. We love the work put forward by these authors and we recommend to check out their work. It is well worth the read.

By living one of our core values, scientific but not heartless, we are experimenting with the ideas from the lean enterprise but looking for ways to incorporate them in an organisational context, as opposed to a startup. While these are not mutually exclusive (they have an intersection like a Venn diagram can) there are some subtle differences that can have a significant impact. Our hypothesis statement for this playbook ends with; *when a software/people fit is found, momentum is gained by the organisation and business agility is increased*.

The Product/Market Fit (PMF) concept was developed and named by Andy Rachleff [1]. Essentially it means that you've found the holy grail, a profitable market with a product that satisfies that market. It sounds simple enough but requires a deep understanding of the market you are targeting and is often measured by whether the market is willing to pay for it. Software/people fit is targeted towards the lean enterprise and once achieved, it means that you've found a software solution to a business problem that fits the people using it. The community of people who rely on the software will not necessarily pay money for it, so the value (or fit) must be measured by the impact it has on business agility.

[1] 12 Things about Product-Market Fit `https://a16z.com/2017/02/18/12-things-about-product-market-fit/`

If business agility is an organisation's ability to respond to change, then how do you create sustainable business growth in a seriously unpredictable world? In the corporate environment, a highly scalable business is one that can maintain or improve profit margins while sales volume increases. The challenge with chasing a highly scalable business is that it often puts profits before people. Imagine instead, a network of teams within a community-centred culture, that embrace organisational values and capabilities and learn through experimentation, in the pursuit of a shared vision that is enabled by codebots and co-creates value for all stakeholders. Which type of company would you rather be a part of? A focus on business agility enables the community of people within the organisation to be more adaptive, creative and resilient in times of unpredictability, with a focus on experimentation to test new ideas.

In the lean enterprise book, the idea of build-measure-learn feedback loops is used for quickly establishing how effective a product, service or idea is. We will go into greater depth on the use of science plus iterations in Chapter 3, however our preference for transforming legacy systems is to use hypothesis statements and the experimental framework, which explores and builds on ideas of the past.

In this book we will discuss the tactic of using science plus iterations as part of an organisation's momentum. But what is momentum and how is it different from velocity? Velocity is defined as the rate of change of position of an object. Momentum is the product of the mass and velocity of an object. We often speak about velocity when we should be referring to momentum, which takes into account the mass we are trying to move. In context of an organisation, small movements made every day by the wider community will create far greater impact than if a few individuals make big leaps of change. This enables continuous modernisation to naturally occur as part of the organisation's momentum.

1.4. The Insanity of Legacy Systems

A legacy system is any software system that is deployed into an organisation. From day one, the software is legacy. It does not matter if the software is some off-the-shelf application or something that is custom built, the day it arrives in the organisation is the day its legacy begins.

Legacy software has been a problem since the beginning of programming. In 1980, Lehman defined the laws of software evolution [20] and identified the sources of evolutionary pressure on software systems and shows why this results in a process of never ending maintenance activity. One of the key points is that it is never ending. This results in the total cost of a software system skewed towards an average of 70% on maintenance and 30% on development.

Without spending the appropriate budget on a legacy system, the decay of the system will fasten but eventually the legacy system becomes such a burden on the organisation that some action must be taken. The action is usually to replace the legacy system with a new software system. But then the cycle continues as the new software system becomes the legacy system and history repeats. This is the insanity of legacy systems.

In Chapter 2, we devote an entire chapter to legacy systems and unpacking the root causes of legacy using the five-whys technique. What is interesting about the root causes is that it generally comes down to either a lack of control or a lack of knowledge. This is an important acknowledgement as our hypothesis for continuous modernisation to have a positive impact on business agility, the root causes must be risk mitigated and built into the solution for legacy systems.

1.5. Knowledge is Power

Have you ever read the book Sapiens [14]? It is the story about homo sapiens from a historical perspective. The author paints a picture of how our species has gone through a number of revolutions including the cognitive, agricultural, and industrial revolutions. Now here is a spoiler alert, basically everything we create as a species is a myth and is only contained within our collective imagination. For example, the concept of money as a system of exchanging goods. The author uses many examples to support this perspective with one example that really stands out and is highly relevant to our discussion here.

A business or an organisation is a legal entity just like individuals are legal entities. An organisation can own property like an individual can. An organisation can take legal action or have legal action taken against it like an individual can. But an organisation can live on well past the original founders and have its own vision, mission, and values that drive the direction it takes. Yet with all these rights and responsibilities how does an organisation keep and retain its knowledge? Individuals use our brains and tools to help us. What does your organisation use?

In a lot of cases, the knowledge of an organisation is wrapped up in the individuals who work within it and this raises a number of risks. An obvious and well-known risk is when an individual leaves the organisation a lot of the knowledge will be lost with them. In this case we try to do a knowledge handover before the individual leaves. Another risk is time wasted, whenever the knowledge contained within a single individual is required by a different individual, time is spent communicating between them. To help mitigate this risk, we try to get people to document their knowledge so that others can read about it or watch a video. In this way, the individual with the knowledge is not interrupted every time someone else wants to attain that knowledge. There has been varying degrees of success using a body of knowledge within organisations and to do it well is a big challenge.

From an academic point of view, a body of knowledge is usually viewed as a tree. Ideas build upon other ideas. For example, in physics it was general relativity that layed the ground work for quantum mechanics. It was then quantum mechanics that lead to the silicon-based transitor and modern computing. The tree of knowledge is a common metaphor used to describe scientific discovery.

Actually, it is rarely that clean cut. The influences on an idea can come from many different directions. In maths a tree is a directed graph, i.e. all of the transitions on the graph go in one direction making a tree. A body of knowledge is more like a graph than a tree. The influences of an idea can be like this with many transitions from other ideas (nodes) in the graph. Another way to think about it is to compare the way a human brain works. The neural networks of a brain are connected and its the collection of neurons that create your knowledge.

Knowledge Management (KM) has a long history and is concerned with how an organisation can make the best use of knowledge to reach its objectives. Overtime, KM has been researched and applied from many difference directions. In Figure 1.4, the layers in the Data, Information, Knowledge, and Wisdom (DIKW) triangle show the relationships between the layers. *Data* is pure and simple facts with no particular organisation; *Information* is structured data that adds meaning within the context; *Knowledge* is the ability to use the information strategically to achieve ones objectives; And *wisdom* is the capacity to choose objectives consistent with values of a larger social context.

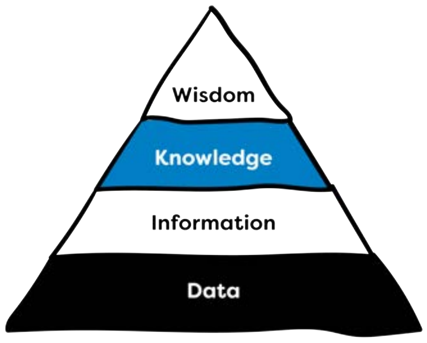


Figure 1.4: Data, Information, Knowledge, and Wisdom

What is very interesting from a continuous modernisation perspective is how do we use KM in way that has a positive effect on an organisations business agility. When we typically build a software system we have lots of conversations (using language) and draw diagrams. We then take this information and capture the knowledge in some sort of requirements specification and hand that over to the development team. The development team then writes the code to satisfy the requirements but there is usually no strong connection between the source code and the knowledge. This is typical software development whereby we have leaked data, information, and knowledge throughout the entire process and left ourselves in a poor position for any further updates. This does not sound wise.

Stepping back and looking at the bigger picture there are some important questions we must ask ourselves. How do we stop leaking knowledge and make the collection of knowledge easier? How do we organise the knowledge so that it is connected and found when needed? How can we collect the knowledge in a way that a codebot understands? The answers to these questions are addressed throughout this book but hopefully these types of questions have sparked your curiosity. As a leader, and we all are no matter our position in an organisation, we must look at knowledge in a new light.

If we know that organisations are a Sapien myth [14], this sheds a unique light on why knowledge is as important to an organisation as it is to us as individuals. An organisation has an existence beyond any individual. This should motivate us to create a body of knowledge. If we do not, our organisation will suffer from amnesia and struggle to learn from the past and teach about the future state it is aiming for. Without sufficient knowledge, we lose control over a situation and this can have a negative impact on the outcome. The organisations that will not just survive, but thrive in the modern era will have knowledge as a big part of their organisational health.

1.6. Bots that Code

Codebots is both a methodology and a set of technologies. It is important to have an understanding of both as it will affect some decisions at the beginning of a legacy migration project. Firstly, the methodology is complementary to your current software development process. It has been designed to be used alongside both iterative and waterfall methodologies. There is no need to change your current software development methodology. Secondly, you will need to choose a codebot to use at the beginning of the project. Each codebot writes to a different technology stack. For example, Springbot writes to a Java server-side using the Spring and Hibernate frameworks in the target application. If this server-side technology stack does not suit your organisation, then there are a number of other public general-purpose bots for C# and LAMP that can be used too. However, if your organisation has specific technology needs beyond these, which can sometimes be the case, then we build private codebots specific to that organisation and technology stack.

In Chapter 4, we take a closer look at the codebots and answer many of the frequently asked questions we receive. To get you started, we often get asked how do you use a codebot? On the platform, there is an intuitive drag-n-drop user interface where you can manage, build, test, and deploy your applications. Like all software systems, there is some input to the codebot, the codebot does some processing, and the output comes out the other side. The output is code that looks like a developer has written it and not a jumble of code generator mess. The codebot writes to a testing target as well. So, upon starting a project you already have good test coverage and a framework for you to start adding in more tests as required.

Codebots can deliver speed to a software project as it does the heavy lifting of the project. Our data shows over 90% of the code base can be written by a codebot and development teams are 8.3 times faster[2]. Codebots can deliver better quality by facilitating test tool development and supporting increased test coverage. Codebots provides reuse by encapsulating common business patterns to be used across multiple projects and teams.

1.7. Future of Work

If this is the first time you are reading this book, then we would like to say a big welcome to the community! We started Codebots with the mission to help our community build better software and we do this by living our values day in and day out. Just in case you skipped over the preface, you can read more about our values there.

It is our vision for humans and codebots to work better together. Whenever a new technology is introduced there are always skeptics, doubt and even fear. We believe that responsible use of technologies has the potential to set humans free from repetitive, mind-numbing activities and opens up new possibilities for the future of work. Machines have repeatedly proven over the course of history that with responsible use, they can work together with humans for our betterment. So far we have demonstrated that both humans and bots can work on successful projects without

[2]Read about the Codebots Field Trials in Appendix B and how these results were obtained.

destroying jobs. There is an unquenchable thirst for software world-wide, and a limited amount of trained professionals to deliver projects, so the Codebots technologies is fulfilling a market demand. It also provides the necessary tools for your local team to compete with offshore development teams and bring operations back on shore.

Codebots are fantastic at finding patterns and doing the heavy lifting on software projects, but it's the humans of an organisation who are able to be adaptive, think creatively and be resilient in the face of technological and cultural change. It requires a move from thinking about organisations as machines, to organisations as communities. This naturally creates an environment where bots and humans can work to their individual strengths in ways we are still discovering, with the purpose of finding software/people fit.

1.8. Summary

As you undertake a legacy migration project, you will face many challenges that the codebots can help with. Codebots is not a silver bullet that can magically migrate a legacy system at the click of a button; it is a methodology and technology set that decreases the risk of failure. In Chapter 5, we present the migration kit. The kit contains a number of activities that will help you find your software/people fit. Some of the activities are purely technical; for example, how to migrate from an old database into a new microservices architecture using the OpenAPI standard but locked down with AAA security. Some of the other activities are related to people; for example, how to manage expectations when you are asked to report on how long a project will take. You must be able to give an answer and most importantly, manage the expectations of the stakeholders.

Eric Evans in his book on Domain-Driven Design (DDD) [8], describes legacy as a ball of mud that oozes out and entangles itself not just with other software products, but within the business processes and people of the organisation itself. By recognising this as a natural byproduct of using software systems means that we can look for strategies and tactics to minimise the impact of legacy software and increase business agility along the way.

We expect our technologies to evolve as we realise our vision of humans and codebots working together, but our mission and values will stay fixed and always remind us of our ’why’. As Simon Sinek [30] says, start with ’the why’ to define your vision that then becomes your company’s North Star or Southern Cross (for those readers in the southern hemisphere). We have an ambitious roadmap with lots of exciting and ground-breaking ideas that we will continue to invite the community to be a part of over time.

2. Legacy Systems

“What is the definition of insanity? Doing the same thing over and over again and expecting a different result.” Albert Einstein

What is a legacy system? Depending on which conference, cafe, meeting room, or pub you are sitting in, you will get many different answers to this question. For the purpose of this book, we define a legacy system as a software application that has been deployed into your organisation. The moment a new software application is introduced it is legacy. It does not matter if it is off-the-shelf, custom built, or some combination. Legacy starts to set in immediately.

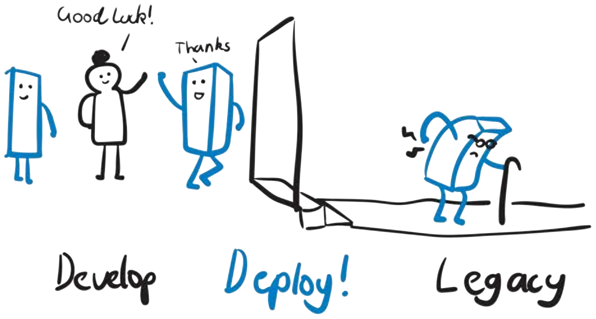


Figure 2.1: The moment new software is introduced, it becomes legacy

Our friends from Gartner recommend seven options for modernising: encapsulate, rehost, replatform, refactor, rearchitect, rebuild, or replace [24]. Our other friends from Cognizant recommend a smaller set of five options: total transformation, gradual replacement, duct tape approach, improve existing, or no system change [28].

These are all very good high-level options and give broad options to consider. To dive down further into the detail on how to actually implement a plan, we first need to find out the root causes. A great technique for this was developed by Sakichi Toyoda and was used within the Toyota Motor Corporation during the evolution of its manufacturing methodologies. It is incredibly simple, when a problem occurs, you drill down to its root cause by asking “why?” five times. So, lets give this a go and see what results we can find.

2.1. Root Causes

Human resources turnover has lead to the new lead developer demanding that the existing software application needs replacing. Why? Because it’s too hard to maintain. Why? Because the developer who maintained it has left the business. Why? Because she inherited it and was frustrated. Why? Because there was really poor documentation. Why? The root cause is because we don’t have access to the knowledge that we need.

The CTO is presenting to the management team and insisting that the learning management system needs replacing. Why? Because it’s a monolithic application. Why? Because it can’t be customised. Why? Because it’s built around processes that are no longer relevant to our business. Why? Because we recently changed our business processes. Why? Because we needed to respond to the market. Why? The root cause is because we don’t have the control over the software that we need.

The business analyst in a large real estate company wants to replace an access database. Why? Because it doesn't do what we need it to do. Why? Because it's not in the cloud? Why? Because it's a local database we built when we were 20 people in one location. Why? Because back then we could collaborate in person. Why? The root cause is because we don't have access to the knowledge that we need.

The entire marketing team is freaking out and wants a new Customer Relationship Management (CRM) tool. Why? Because it's an off-the shelf product and it's no longer supported. Why? Because the vendor no longer exists. Why? Because they were acquired and shut down. Why? Because they were competition in the market. Why? The root cause is because we no longer have control over the software that we need.

That was just four scenarios that we applied the five-whys technique to. As you can see, the root causes surrounding lack of control or lack of knowledge come up all too often. In each scenario, the cause of the legacy system can start from lots of different locations, but they all have a negative impact on your organisation's business agility.

So, how has the world been dealing with legacy systems so far? Besides burying our heads in the sand, we usually either *rewrite* the application, or look to buy something *off-the-shelf*, but all options have many serious pitfalls.

Off-the-shelf systems are brilliant if you can find the right fit. There are no long development cycles to wait for software engineers. Any bugs found in the application are the responsibility of the vendor. And if the licence model is favourable, the total cost of ownership can be better. However, each of these advantages also has a dark-side whereby you can get tied in a knot. Development cycles can become longer if any customisations do not fit neatly with the application. Bug fixes can take an unacceptable length of time as the vendor deals with the enormity of their back log. And the total cost of ownership could blow out as circumstances change and you become vendor locked and cannot manoeuvre. Sometimes we just live with this because the alternative is a lot worse.

On the other hand, we could look at rewriting a legacy system. Writing a new application is fun and working on a greenfield application is hands down the best time for a software team. We all know this. So, if you ask a software team how to deal with a legacy system, they will recommend that you rewrite and you must use the latest and greatest technology that all the other companies are using, and if you don't, you will be at risk to your competitors. There is some truth to this, but the moment we rewrite an application it becomes the legacy system of tomorrow and history repeats!

So, how do we break the cycle of legacy systems? Well, the short answer is we must embrace legacy systems and move into a mode of continuous modernisation. In highly complex and dynamic ecosystems, it is the application of strategies and tactics to minimise risks and play to our strengths that tip the odds in our favour for a successful outcome.

Before this can happen we must recognise some inescapable truths to which there are no silver bullets. But for the more experienced professional, we know that once a risk is acknowledged, then mitigation can be put in place to help manage it[1]. So, what are the inescapable truths?

2.2. Inescapable Truths

The first is an old phenomenon … *garbage in, garbage out*. For any process that has inputs and outputs (and we are not just talking about software applications), if you put garbage into it, you will get garbage out of it (whatever it is). There is no escaping this. So, if you try to take a legacy system and run it through some sort of migration tool, you will get all the garbage on the other side as output. This is especially true in our experience when attempting migration from source-code, i.e. if you attempt a source-code migration from an older programming language to a more modern one.

This inescapable truth leaves you two broad options. You can either clean the garbage before it goes in, or try to clean the garbage as it comes out. Whichever way you choose, you will run into the second inescapable truth, *knowlegde is always missing*. In your quest

[1]Have a look in Appendix E for our top risks.

to gain visibility and find the required knowledge about the legacy system, you will likely come across end users, project managers, scrum masters, software engineers, designers, and all sorts of people that may have influenced the system (this is usually the role of a business analyst). But there will be holes (sometimes huge ones!) that people simply can't fill with answers as no one knows where the documentation is ... or they didn't have time to do it. This means that knowledge is always missing about a legacy system and something that cannot be avoided.

This lack of knowledge may result in teams being afraid to make changes, which leads to the next inescapable truth; this is as much a *people* problem as it is a technical problem. Once you have solved the garbage in, garbage out problem, if you go ahead and build a great software application without involving the people who will be using the application, the project will be doomed to failure. The truth is that people are complex and at times resistant to change. Who can blame them? Just think about how many times big projects that will revolutionise an organisation have been announced and then just fade away. The key is to identify the community of people who are most invested or opposed to the new system, and take them on the journey. We will explain how community can be used as a tactic later in the book but be ready to prioritise the people and don't underestimate how long this can take!

These are the inescapable truths that you will come across for any legacy migration project. There will be significant effort to gather the knowledge around the legacy system to make sure that the garbage is not carried through to the new system. You will need to involve the people to help with this process and take them on the journey, so ultimately the new system is embraced and has a positive impact on the organisation.

From the inescapable truths we know that we are going to be dealing with a lot of pain. This is not going to be easy. However, our innate satisfaction as homo sapiens is proportional to the relative level of difficulty we can master. This has resulted in an interesting shift in the mindsets of our software teams. Previously, when legacy system projects were seen as undesirable, brownfield projects, no one would want to take them on. Especially with such a high rate of failure. Now, they are seen as a challenge worth taking on. Greenfield projects are easy in comparison.

Legacy systems come in all shapes and sizes. In the next chapter, we are going to dive in deep and talk about some strategies and tactics. But, before we do that, let’s get your creative juices flowing and discuss some scenarios.

Legacy systems become entangled within the organisation. They become tightly coupled with other software systems. They become the normal for how people expect to use the system, possibly ending up in scenarios in which people have to manually copy and paste data between applications to stitch up a business process. It is common sense that when approaching a legacy migration project not to use a big bang approach i.e., do not to migrate the whole system at once as it is typically a bad idea. But there are scenarios where the legacy system's entanglement is low and it is fairly isolated. In these circumstances it could be possible to migrate the legacy system in a project, instead of being a big bang, we call these firecrackers! The firecracker migration pattern is discussed in Section 2.3.

Alternate to the firecracker we use a divide-and-conquer migration pattern. This pattern is used when the legacy system is too large to complete in a single step using the firecracker migration pattern. The divide-and-conquer migration pattern is discussed in more detail in Section 2.4. Figure 2.2 shows a flow chart of the decision making process for which migration pattern to use. You will notice that both migration patterns ask for a Structured Query Language (SQL) compliant schema for the migration. This is the best starting point for a migration project as it leads to a data migration path from the old legacy system across to the new application. However, if you are not migrating from a legacy system backed by a database, don't stress as you can substitute the database for whatever technology you are using. The steps will be the same but the implementation will be different.

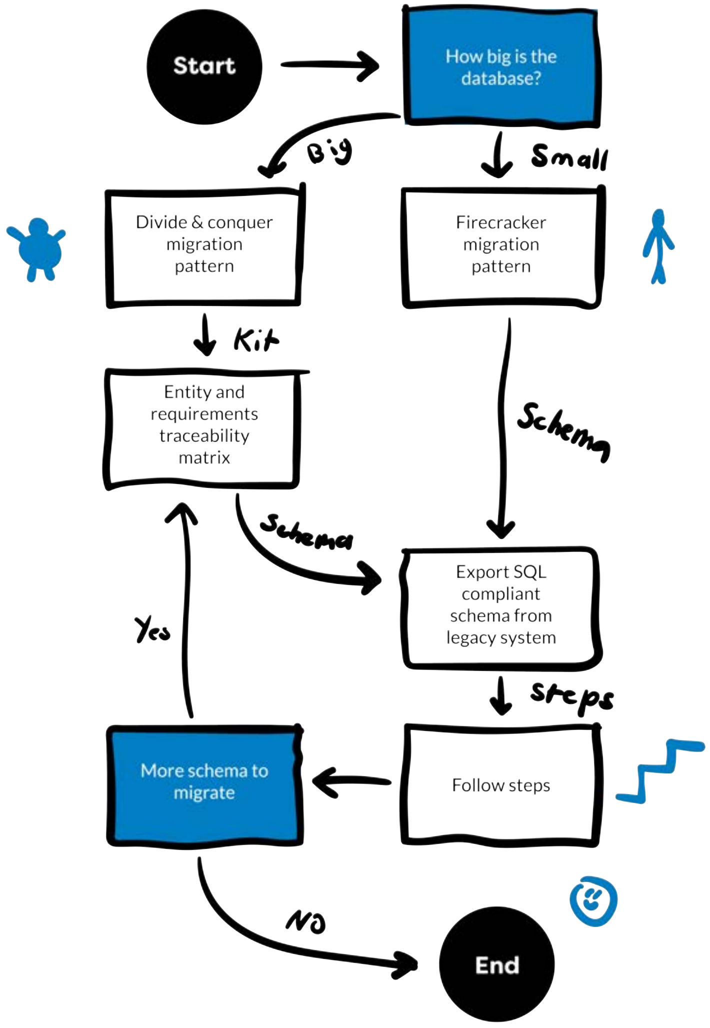

Start
How big is the database?
Big
Small
Divide & conquer migration pattern
Firecracker migration pattern
Kit
Entity and requirements traceability matrix
Schema
Schema
Export SQL compliant schema from legacy system
Steps
Yes
More schema to migrate
Follow steps
No
End

Figure 2.2: Database migration workflow

2.3. Firecracker migration pattern

This is the first of two migration patterns we are going to discuss in this chapter. For smaller projects, firecrackers can be used as the size of the project is small enough to do in a reasonable amount of time. In Figure 2.3, we show a conceptual picture of a firecracker migration pattern. The approach is to gather knowledge about the old application (bottom right) and then build a new application (top right) from this.

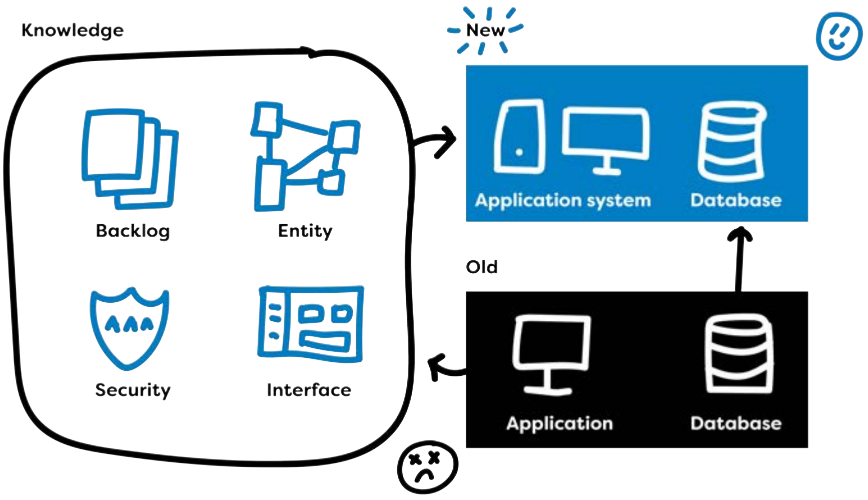


Figure 2.3: Firecracker migration pattern

The following points describe the broad steps of the firecracker migration pattern:

- Document the requirements of the old application to create a backlog for the new application. Remove old requirements that are no longer needed [2].
- Reverse engineer the schema from the old database. Clean up the schema to remove any garbage.
- Design the User Interface (UI) to satisfy the requirements from the backlog,
- Develop and test the new application.
- Deploy and migrate the data. Since you started with the old schema, a migration path is possible.

As a broad approach to migrating legacy systems, the firecracker migration pattern works well for smallish and fairly isolated legacy systems. But something is not quite right, haven't we just replaced the old with the new and gone back to square one? We have, though we have missed one very important step. We are going to make that knowledge about the legacy system as important as the system itself. We are also going to take it one step further: we are going to have a codebot take that knowledge to write the vast majority of the new system. So, the next time we are updating the old legacy system, we are already at the point in which we are ready

[2]See Section 5.2 of the migration kit on reverse engineering requirements.

to go and can make some changes. Later in this book we are going to discuss in more depth the role of models to sustain knowledge (Section 3.5). We are also going to look at how these codebots work and the benefits they bring to your organisation (Chapter 4).

2.4. Divide-and-Conquer Migration Pattern

The firecracker migration pattern is a tried and tested approach to migrating legacy systems that are smallish and fairly isolated. However, let's notch this up with a more complex scenario! For the non-technical readers, this section is heavier reading so you may want to skip over here and possibly come back later when you are seeking some finer level details.

It is quite common to come across a legacy system that is so large that it cannot be replaced in a single project. It would be impossible to do so and any attempt would be to the detriment of the organisation. So, we must divide-and-conquer!

Take some time now to think about how you would do this. It can be mind-boggling. How do you deal with having an old and a new system live at the same time? How do you ensure that the people using it have everything that they need? How do you deal with data integrity and make sure you are not breaking other systems up-stream or down-stream? These are all very good questions and we are going to present a divide-and-conquer migration pattern as shown in Figure 2.4.

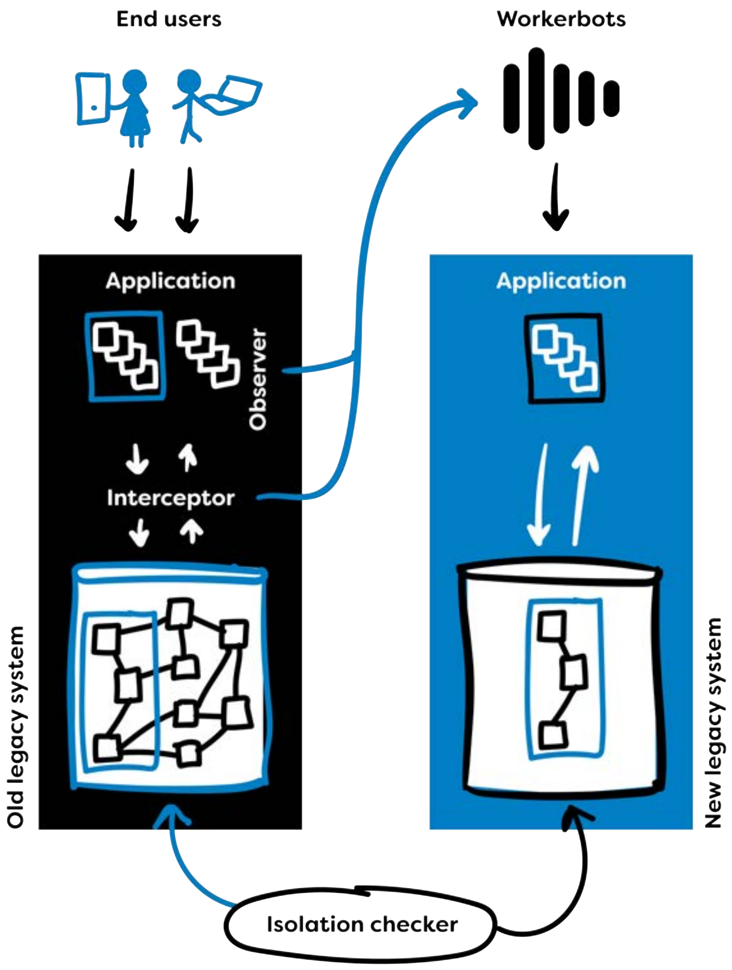


Figure 2.4: Divide-and-conquer migration pattern

This figure is more like a deployment diagram. The old legacy system is on the left and the new legacy system is on the right. Each legacy system is split into an application (top) and a database (bottom). The humans on the left are the end users of the old application and the workerbot on the right will simulate the humans during the transition phase. The following points describe the broad steps of the divide-and-conquer migration pattern:

- Follow the steps from the firecracker migration pattern i.e., reverse engineer the requirements and design the new application. But only a subset of the requirements.
- If possible, add an interceptor between the old application and its database to forward all requests to the workerbot.
- If possible, add an observer to the legacy application that is able to forward other request data to the workerbot.
- Since all projects are different, think of different observers and interceptors you can use to send to the workerbot.
- Develop the workerbot to interpret the requests from the observers and interceptors to mimic the same requirement but in the new application.
- Use an isolation checker to ensure the data integrity of the new application. It's possible that some requirements have been missed and its important to do a delta between the databases and make sure it's what you expect.

It will take significant effort to build the framework around the old legacy system. The observer, interceptor, and workerbot will take time to get right. But once you get them going, you will have a path to continue the divide-and-conquer approach on the old legacy system. The seeds you sow today will yield the crop of tomorrow!

Before we get too carried away, there are some other scenarios worth discussing as well. What if the old legacy system has input from an up-stream system? Or, what if the old legacy system outputs to a down-stream system? For any input that comes into the old application, the approach is to use interceptors and observers to pass it onto the new application. And for any outputs from the old application, the approach is to have the new application do the same, but to a mock[3]. Then use an isolation checker to ensure that the expected output from the old application compares to the mock.

The last complex scenario for discussion is when the legacy system is entangled with other systems. A common type of entanglement is a shared database, i.e. when two applications share the same database. This is probably one of the worst design decisions that we have made in the software industry. As an uncontrolled interface there is no telling how deep that rabbit hole might go. The best approach to deal with this scenario is to treat the two applications as if they were one application. So, when you are

[3]Mocks are common in testing to mimic another system without it actually being there.

following the divide-and-conquer migration pattern, you may need to build multiple observers and interceptors on both applications so that the isolation checker will pass. You can see the importance of the isolation checker here as it will alert you if there is a third application using the database that noone even knows about.

2.5. Summary

In summary, we have taken a close look at legacy systems. For the purpose of this book, we define a legacy system as any software application the moment it is deployed inside the organisation. It can be custom built, off-the-shelf, or anything in between. The insanity of legacy systems is that once we replace them with a new application, that application becomes the legacy system of tomorrow! So, how to we break the cycle? The smart approach is to set yourself up not just for the first legacy migration, but all the subsequent ones after that. The trick is to make the knowledge about the legacy system as important as the system itself. This will minimise the knowledge loss across the organisation and enables a codebot to write most of the new system for you, based on that knowledge. Is that super cool or what?!

3. Strategy and Tactics

"Success is not final, failure is not fatal. It is courage to continue that counts." Winston Churchill

We like to know the big picture. If you can visualise what you are trying to achieve, then you can show initiative and make decisions that do not require someone to give you direct approval. Knowing the big picture is the strategy, how you achieve the strategy are the tactics.

A strategy is the overarching plan. Tactics are specific actions undertaken as part of the strategy.

To help build a visual picture of our proposed strategy, we are going to use military strategies as an analogy. Military strategies take many forms and are always dependent on the terrain and units involved on both sides. We are not going to go that deep but there are some well-known strategies worth exploring.

A frontal assault is typically a last resort as you would be subjecting yourself to the maximum defensive power of the enemy. If the flanks (sides) of the enemy are an option, this can be a much better way to attack (see Figure 3.1).

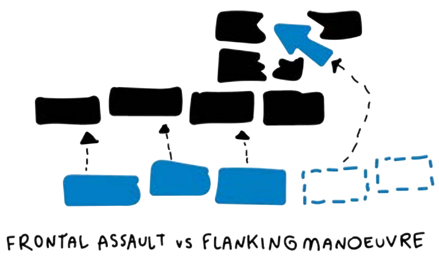


Figure 3.1: Frontal assault vs flanking manoeuvres

If there is no option for a flanking manoeuvre and a frontal assault is the only choice, a smarter strategy would be to maximise your offensive power on only part of the front to penetrate through the enemy line. Once through, you can then attack from behind the enemy where they are weaker as their defences have been setup facing out. This is a well-known strategy for breaching the enemy line and attacking out over several phases. In the armoured and motorised warfare, it is sometimes referred to as a Blitzrieg strategy. *Phase 1* is to penetrate the enemy line, *phase 2* is to regroup and recognisance, and *phase 3* is to attack other enemy positions.

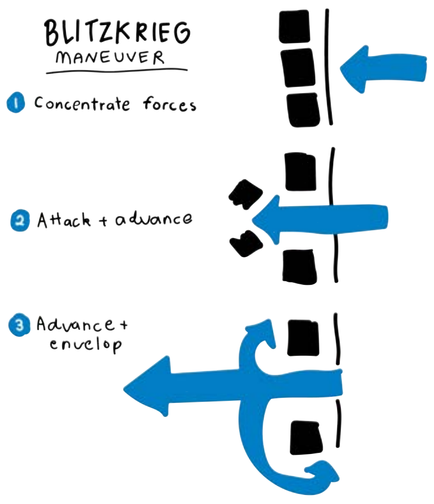


Figure 3.2: Blitzrieg

So, why all this talk of military strategies and tactics? The analogy goes like this; the enemy is your software ecosystem. If you have lost control of your software ecosystem and legacy systems have taken root, then you are burning cash and not able to appropriately support the business to carry out its purpose. In reality, your software does sometimes feel like this. Once you have relative control of your software ecosystem, you will be able to manoeuvre to meet changing business needs. So, it is time to apply some strategies to modernising the ecosystem.

A complex software ecosystem has no flanks i.e., no matter which direction you approach it from you are always faced with the same problems around legacy and lack of manoeuvrability. A frontal assault would be folly, as there are simply too many applications in a complex software ecosystem to tackle at once. We have worked in environments where there are not hundreds, but thousands of different applications.

This implies a good strategy for us to explore is the Blitzrieg. We want to maximise our efforts by concentrating our forces. As seen in Figure 3.3, *Phase 1* is to modernise a legacy system, *phase 2* is to map and connect to other systems, and *phase 3* is to build new applications and modernise other legacy systems.

The hardest part about getting started can be knowing where to begin. Standing at the base of a mountain looking up at the peak can be intimidating and while it's important to think big, you want to move past the initial overwhelm and start small. One foot after

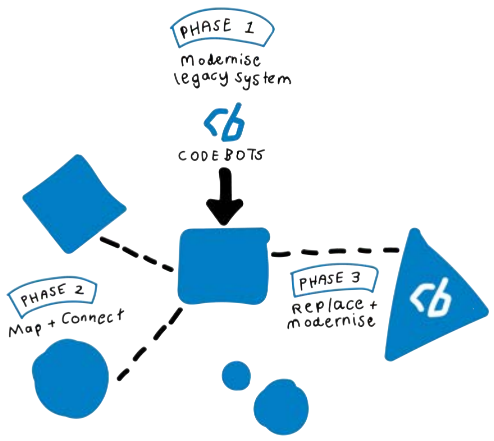


Figure 3.3: Three phase strategy

the other, you begin by identifying a problem that needs to be solved and rallying the support of a core team of people. You don't have to know all the answers upfront; you just need to agree why you're doing it and start the learning.

Once you have your core team engaged and have proven your playbook of strategies and tactics on smaller projects, you will approach a magic moment when the momentum you've built reaches a threshold. In Malcolm Gladwell's book, The Tipping Point, he puts forward the concept that small changes lead to making a big difference that can start an epidemic. This is exactly the kind of change that a digital transformation can spark. Starting with a small, focused team who use the learnings they've made as a guide to find a repeatable process that will gain traction in the business, or as Gladwell refers to it as the stickiness factor. This repeatable stickiness becomes the mechanism to incorporate learnings into subsequent projects.

Now is the time to scale up to larger teams but remember, think big but start small! A tight team of 4 to 6 empowered individuals is enough to get your digital transformation underway. Building on the momentum and history of small successes, you are able to rally the support necessary to execute on the bigger vision of continuous modernisation. This sets the organisation up to take advantage of changing market conditions, deliver value on business outcomes and impact their business agility.

With a strategy planned out, let's get into some of the tactics and look at the detail. The five tactics:

1. Use science to drive innovation at a pace in step with the organisational rhythm.

2. Enable the community around your organisation. A strong community builds a defendable moat around your organisation.

3. Build small loosely coupled microservice applications emphasising a separation of concerns.

4. Increase the visibility and control of the entire software ecosystem by lifting the fog of war.

5. Use models to capture knowledge and enable control through the use of codebots to do the heavy lifting on software projects.

3.1. Science + Iterations

One of the most important concepts to recognise when planning out strategies and tactics is to acknowledge time. According to Einstein, time is relative to the observer's motion and it is not constant. It is the speed of light that is constant. But for us mere mortals that live our daily lives on this planet, time does feel like a constant and it will continue to roll forward at the same rate no matter what we do. Since the passing of time is inevitable, it would be advantageous for us to make it an ally of our strategies and tactics. Can we use the concept of time (without knowing the future state of a project) to help shape and increase the chances of a successful project? This may seem a strange question, but in our experience with large projects, once the ball is rolling and the project is underway, it is impossible to impose a pure top-down structure over the project to ensure its success. This naturally leads to questions around enabling a more bottom-up approach. Something that enables the team to have freedom to be creative and solve problems, but to have bumper lanes to keep them on the path.

The tactic is to concurrently use science plus iterations to enable continuous modernisation to naturally occur as part of the organisation's momentum. In this section, we will delve into what makes good science and the use of iterations as part of an organisation's operating rhythm.

Science is the great enabler and it really started to accelerate during the renaissance period of the 1600s. At this time, humans began to realise that they didn't know all the answers and some of the fundamental things that they thought were true, were false. Galileo gathered evidence about a theory that Copernicus had proposed and it showed the earth was not the centre of the solar system. This was a huge and fundamental shift in the thinking of humans and what it demonstrated was that maybe we simply do not know. During the following centuries it became the norm to ask fundamental questions and what we believe to be the coolest thing about science, is that it's ok not to know the answers.

We love science so much that we have made it one of our values! But some people do not love science. Usually the reason for this is flashbacks from school where you are trying to work out some complex maths problem and it feels like your brain is on fire and you end up feeling stupid because you can't find the solution. We have all been there, but that is not science; maths is one of many tools used by scientists and science is a whole lot more. Science is powerful and if you are not using it already, it is time to get in the mindset.

How does science work? First, you ask a question that you do not know the answer for. And it is ok not to know and other scientists respect this, which creates a great culture when talking science. After you have proposed your question then you go about finding the answer. That's it! But to sound more scientific the question is your hypothesis and finding out the answer is your research where you run experiments to collect evidence. The result of the experiments can be both positive, negative or inconclusive. Sometimes a negative or inconclusive result can be as revealing as a positive result. So again, this creates a great scientific culture, as it is not only positive results that are respected; it is the contribution to the body of knowledge that counts.

Let's look at a quick example. The hypothesis for this book is that business agility can be impacted by continuously modernising legacy systems. To gather evidence we are going to help organisations run legacy migrations projects and measure the impact this has on their business agility. Throughout the process, we are going to run many experiments and subsequently update the migration kit presented in Chapter 5 with the results. As we get more results we are going to publish new versions of this book and resend it to the community to ensure everyone is kept up to date.

The key to good science is designing good experiments. A poorly designed experiment will lead to no results as no meaning can be derived from the evidence and data. A good way to design an experiment is to reverse engineer the experiment from what the final report should look like. Imagine that you are presenting the report to someone and you have some cool graphs and insight to show them. What is on these graphs? And what are the insights? If you start with this in mind, you can work backwards to discover what experiments and data are needed. Maybe you need to design a survey to feed in some data. Maybe you need your team to collect data as they do some activities. Whatever it is, start with what the final report looks like and work backwards from there.

Usually it takes multiple experiments to gather evidence for a hypothesis. So, how do you get your people doing experiments in an organisation where everyone is already busy? This can feel like a real challenge. Firstly, we have made science one of our core values[1]. We are always talking science and experiments. We have included our experimental framework in Section 5.3, which you can test out as part of the migration kit. Secondly, you need to run the experiments concurrently to the operating rhythm of the organisation. And who is demonstrating the best operating rhythm in organisations today?

For about two decades now, software engineering teams have been experimenting with building software systems using an Agile approach. The Agile manifesto outlined an agenda for how teams trade off and prefer certain things over others. The inception was brilliant; it did not mandate the processes but the mindset. There have been many flavours of Agile invented but the truth is, most companies come up with a hybrid approach that best suits their own circumstances. However, there is a common thread throughout the vast majority of them … sprints.

In summary, a sprint is a short cycle whereby value is delivered in each iteration throughout the project. This is in

[1]Scientific but not heartless.

contrast with traditional waterfall methods that are much longer and deliver only at the end of the project. We dislike the term sprint as it implies we are sprinting like a 100m race, but sprinting is not sustainable over long periods of time. We prefer to call them iterations and talk about sustainable pace by demonstrating urgency, but not being rushed.

Agile has proven quite popular as it can reduce the risk in software projects when the team is fluent and well-rehearsed in their craft. Whilst software engineers cannot claim to be the inventors of this mind set, Agile is beginning to influence the wider business to consider the role of using iterations for more than just software projects. For example, the Lean startup methodology, uses build-measure-learn cycles to gain traction for a product/market fit. Another way to think about it, is to think of an iteration like a heart beat. If you are doing long waterfall projects, the heart beat will be slow and so will your business. If you are able to do shorter iterations for projects, the heart beat will be quicker and this will give your business a greater number of opportunities to evolve.

3.2. Community

Right from the start, our company mission for Codebots was to help our community build better software.

Did you know that the original definition of the word 'company', is derived from the Latin words 'com', meaning together, and 'panis', meaning bread? Merchants would come together, share stories, break bread and do business. They were communities! But during the industrial revolution when the priority was mass production and we moved to a centralised business model, companies started to behave more like machines than communities. Each employee was a small link in a chain, optimised for efficiency but with little opportunity for individual creativity and contribution.

Today, we have an unprecedented collision of new technology, AI, VR, blockchain, business agility and the list goes on. We believe that these awesome technologies have the power to set humans free from the repetitive factory mindset. By harnessing the power of codebots, our community can become creators and inventors again, letting the technology do the heavy lifting. But this can only happen if we shift away from thinking about organisations as machines, to organisations as communities.

This tactic is based on the core idea that every person that interacts with your organisation is a member of your community. The metaphor of an onion [10] can be used to visualise the different layers of membership in any business or for any given project as illustrated in Figure 3.4. The people at the core are those whose contribution is the greatest and who are the most invested in the mission, vision and values (founders, investors and leaders of the company). They influence and inspire the next layer of team leaders and employees, who in turn influence and inspire loyal partners and customers, who then spread the word to our fans and followers.

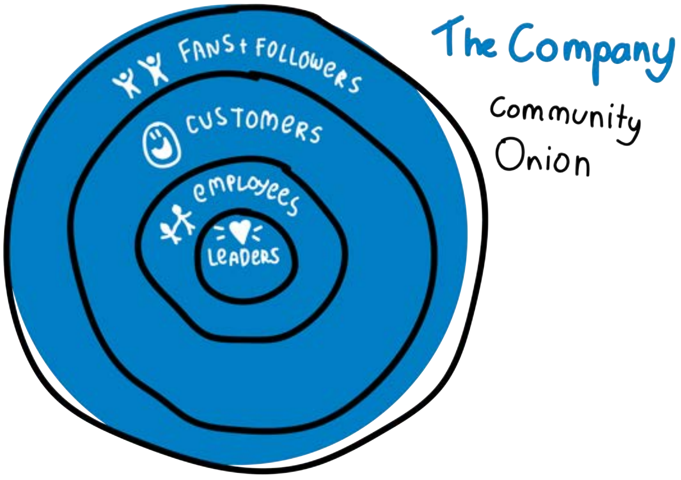


Figure 3.4: Community onion

Rather than differentiating our employees from our customers, we connect the dots and consider it a single community experience, with different levels of contributions and experiences within it. When you think about it in this manner, you are able to set and shift the culture from the core. As long as the most invested community members are aligned on the messaging of the mission and their behaviour consistently reflects the values, it will be echoed and embraced as you move outward through the layers. In Section 5.4 we will explain how to apply the community onion to a migration project as an activity within the migration kit.

We use tactic *science + iterations* to experiment with programs that move individuals on the journey from awareness right through to brand ambassador. By using *community* as a tactic each member has the potential to become an advocate, not just of your technology (which can always be copied) but of what you stand for, and they identify with and take ownership of that vision. This builds an invisible but impenetrable moat of good will, advocacy and genuine affection around your business. Your community does not just use your product, they are your product, and they will promote it to their friends and colleagues, they will tap their networks to recruit like-minded employees and they will voice the innovative ideas that are key to driving a culture of continuous modernisation and finding true software/people fit in the organisation.

As business management author, Patrick Lencioni [21] says, if you can get all the people in an organisation rowing in the same direction, you can dominate any industry, in any market, against any competition, at any time. So why is it, that the one variable that is often forgotten is the people?

In 2014, MIT Sloan Management Review, in collaboration with Deloitte[23] surveyed more than 4,800 business executives, managers and analysts from around the world and found that many companies focus on technologies, without investing in organisational capabilities that ensure their impact. They concluded that many failures are because organisations did not change the mindsets and processes, or build cultures that foster change.

The community tactic can be applied to any digital transformation project. It begins by ensuring that those at the centre of the onion are the most invested in making the change e.g., CEO, CIO, project owners, are aligned and rowing in the same direction on the core message and values. The next step is to take the team on the journey, by getting the next layer of the onion being the key stakeholders to understand and believe in the mission of the project. Then they become the project ambassadors and will rally the support of the outer layers of the team, who are ultimately the end users of the software.

In this book we talk about the modernising of legacy systems being both a technical and a cultural problem. By keeping people at the centre of your strategy and having everyone rowing in the same direction, it is possible to create the momentum and cultural shift required for continuous modernisation by making a software/people fit.

3.3. Microservices

A well-defined *interface* goes a long way to making great software. One advantage of an interface is that it allows developers to seperate the concerns of a program by modularising or making components that take care of something. So, when another developer comes along to use the code they need not be concerned about what happens behind the interface but what the interface provides. The interface will have some input via a request and some output via a response. In modern web-based systems, this is usually done via a web service of some description.

Microservices is an architectural style that was first discussed at a workshop of software architects in 2011. The following year the participants decided on microservices as the most appropriate name. Microservices is a broad term used for different characteristics of modern architectures that were discussed at the workshop. They are summarised by Lewis and Fowler [22] as:

- Componentisation via Services,
- Organised around business capabilities,
- Products not projects,
- Smart endpoints and dumb pipes,
- Decentralised governance,
- Decentralised data management
- Infrastructure automation,
- Design for failure, and
- Evolutionary design.

This list is a mixture of both technical and organisational characteristics. It is a big list and each item can be a book in itself; actually, there are some great books on these worth a read [25]. For the purposes of this book, we are going to simplify the definition of microservices to be: microservices are composed of small independent services that communicate over well-defined APIs. To visualise the importance of what this means, we have provided a diagram in Figure 3.5. The traditional monolith on the left is large and very difficult to change. It is possible to customise a monolith but history has shown that this is costly to update and hard to get a good fit. A lot of monoliths come with everything and the kitchen sink and the organisation usally only ends up using a portion of what is available.

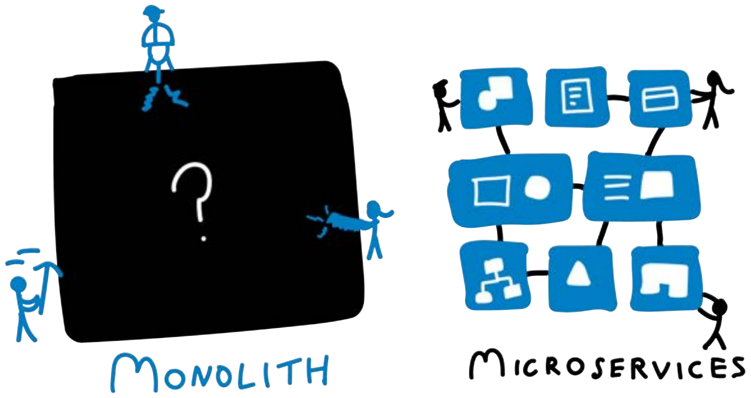


Figure 3.5: Microservices architecture

By breaking a monolith up into smaller microservices, developers can work on and update smaller parts of the overall project as they are loosely coupled with other microservices. This is ine of the biggest attractions on why microservices have gained popularity. For the non-techinical readers you can skip the rest of this section and go onto the fog of war in Section 3.4. We are about to get a bit nerdy and talk about the use of software patterns in the target application. It's safe to skip forward from here as you should now have a broad understanding of why microservices can be more benficial than a monolith.

All of our codebots use a similiar approach to microservices and the architecture represents best practice. Some of the codebots have some changes due to the frameworks they use, such as Spring or the Entity framework, but they generally follow this approach. The server-side is MVC-based with a service-tier and datastore-tier. The controller-tier exposes web services as endpoints. The business logic can be performed in-memory in the service-tier, or using complex SQL in the database-tier and possibly stored procedures as required. The client-side is also MVC-based leaning towards a more thick client with the option to do business logic in JavaScript too. The thick client approach can lean towards hybrid mobile-apps with an offline capability opposed to having the server-side do all the business logic work. The *Double MVC* architecture is a flexible approach to microservices.

Are microservices the future of software architectures? In the context of legacy systems, they are showing positive results as changes to software are achievable within a reasonable amount of time. There are a few pitfalls to watch out for like performance, fail-over, security, enivironments, deployment pipelines, and the like. But we are recommending the use of microservices as a skilled team can make sure these potential pitfalls are addressed.

3.4. Fog of War

The fog of war is used in strategy computer games to hide part of the map from the player. The player cannot see or lift the fog of war until they have explored that area. And in some games, unless there is an active unit within the area, the fog resettles and the player loses visibility again. The idea is that this part of the game is hidden from the player and it makes it more difficult to know what's going on. The best players lift the fog of war on their strategic outcomes and increase their chances of winning the game.

This analogy can be used with your software ecosystem as there are many occasions where there is a fog of war over what is actually going on in the organisation. For example, as a legacy system gets older, the technical debt increases and the ability to maintain or improve it gets harder. This creates a fog around the legacy system as it is harder to know what time and resources are needed to make changes.

A different example of the fog of war is *shadow IT* … the dreaded shadow IT. Shadow IT is all of the applications, both off-the-shelf or custom-built, that are introduced into the ecosystem by lone wolf employees that are showing a little too much initiative. So, unless you have complete knowledge of every bit of software in your ecosystem (and I suggest that might be impossible), you are dealing with a fog of war. The question is, how do you lift it in a meaningful and sustainable way?

While doing research for this book, we interviewed Gary Buck[2] on this exact question. Most organisations will do an audit and create a map of the eco-system, which will lift the fog of war. The problem is that it’s just a brief moment in time. The map from the audit becomes dated and it loses its usefulness and it is rarely revisited or maintained. This means that the fog resettles back over the software landscape and you lose visibility, which has a negative impact on your business agility.

In a strategy computer game, there are a number of ways to contend with the fog of war. Ideally, you would place an active unit in the area of interest so that the fog of war is permanently lifted. Another tactic, is to send out scouts to patrol the area at regular intervals, but this can be a pain if the player cannot automate the task as they will have to devote time to send them out on the patrol. Well, it turns out that as part of the 3-phase strategy described at the beginning of this chapter, we might just have a way to run some similar tactics for our software ecosystem.

In phase 1 of the strategy, we will be migrating a legacy system across to a new microservices architecture. During phase 2, we are going to connect the new application via the services to other applications and potentially create new applications as well. Then in phase 3, we are going to migrate more of the legacy systems to a microservices architecture. This 3 phase strategy is an

[2]Gary was the CIO of BHP for 20 years and is a very experienced practitioner. We recommend checking out his book on the first 100 days of being a CIO [**?**].

excellent opportunity not only to lift the fog, but to place active units to ensure that it is harder for the fog to resettle over the new applications. Without spoiling some of the content for the next section on modelling, and to use the anology so far, the model is your active unit.

If we passed a developer 1.2 million lines of code from a legacy system and asked a developer to modify it, they would most likely look at us like we were insane. As depicted in Figure 3.6, we pretty much just passed them a locked safe without any visibility or instructions to what is inside. This is known as a black-box and brings up all sorts of twitches and flashbacks in software teams.

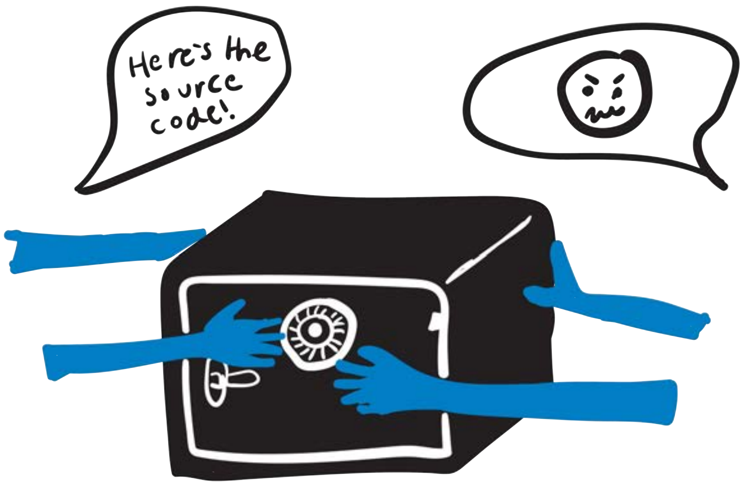


Figure 3.6: The joys of black-box code

If we passed a developer 1.2 millions lines of code with some documentation, they could attempt some changes but history has shown us that it will likely fail or run over time and budget. If we passed a developer 1.2 million lines of code with the requirements backlog, user interface designs, the database schematic, and all the reasoning behind the application and architecture with a connected traceability matrix to tests, they would ask for a little time to study it, but the likelihood that they could meaningfully update the code has improved significantly.

Legacy systems are pretty much this. They are a black-box where the fog has encompassed them completely and cannot be lifted. With any extra knowledge about the application, the legacy system is becoming a white-box and the fog is being lifted. Now here is the trick: the model must become as important as the source code. So, when the model and the source code are presented in combination, humans can better understand and therefore they have better knowledge. They are regaining control of the legacy system through the knowledge that surrounds it and making the legacy application more of a white-box.

3.5. Modelling

Models in software engineering are one of the most under utilised and misunderstood technologies available today. Our industry has a lot of hype around Artificial Intelligence (AI) with Machine Learning (ML) and Natural Language Processing (NLP) showing some amazing results. Other technologies such as Virtual Reality (VR) and Augmented Reality (AR) are also getting a lot of airtime but the little known area of Model-Driven Engineering (MDE)[3] is not getting much attention at all. Yet, in our humble opinion, it is the most powerful technology of them all and has the ability to be applied to any industry, sector, or solution and create an order of magnitude improvement.

Before we start discussing the use of models, let’s quickly address why MDE has received such little attention. Like AI, the concepts of MDE have been around in some form or another since the dawn of the modern computer. Yet, hardly anyone has heard of it. Why? In die-hard modelling circles you will get different passionate responses, but we are going to put forward our thoughts that will hopefully shed some light on the situation.

We believe that academia has clouded the use of models with so much complexity, that students of software engineering and others simply don’t understand it enough to be able to apply it well. And like high school where bad teachers can turn students from subjects, the use of models has likewise been relegated to the back of the queue. It has long been taught that models are only good for critical systems like aircraft control systems or other safety critical

[3]Don’t get too hung up on the acronym used. You will come across other variations in the literature like Model-Driven Software Development (MDSD) [33] and Model-Based Testing (MBT) [32], but they all essentially just mean the use of models in software engineering.

systems. This is not true. We must expand the horizon and look at the use of models further than this.

By the end of this section, the use of models will be demystified. It is the goal of this section to provide you with a good understanding of how the use of models not only works from a technical perspective, but how it can be applied in real-world projects. So, as we journey deeper into our understanding, you will be need to enter with an open mind. We are going to present 3 laws to help your understanding and once you know them, you can make informed decisions around the use of models with the best of them.

What is a model? A model is a simplification or approximation of reality and hence will not reflect all of reality. The important word here is *all*, the model cannot exactly represent reality, and if it does, it is not longer a model. This is known as Bonini's Paradox [3], as a model of a complex system becomes more complete, it becomes less understandable.

The paradox means that there will always be a gap between the model and reality. This is the first law, a model is incomplete as it cannot exactly match the thing it is modelling. How this gap is managed is a key differentiator between different software platforms. Later in this section, we will show you the implications of this from both a business model and technical perspective.

The second law surrounds the meta-model. All hail the meta-model! The meta-model describes what can be found in the model. Once you control the meta-model, you have the power to create any type of model imaginable. The meta-model is the layer below the model. For example, in physics, an apple is made up from atoms and there are 94 naturally occurring elements in the periodic table. Each element in an atom is made up from protons, neutrons, and electrons. The protons and neutrons are made from quarks. And finally, string theorists believe the quarks and electrons are made from tiny strings.

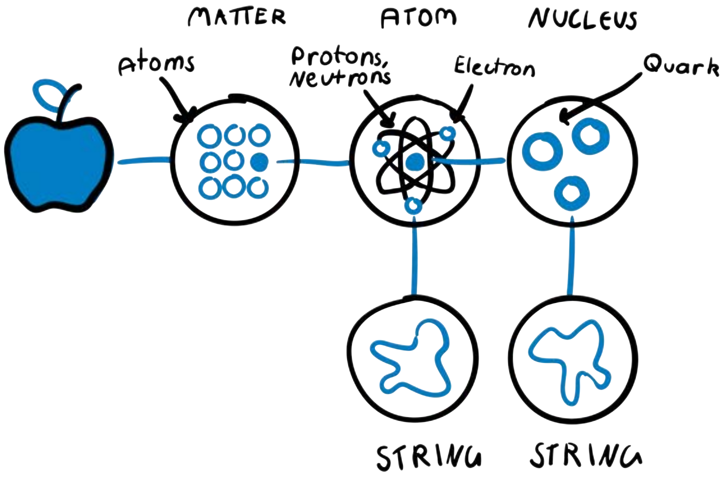


Figure 3.7: Layers in the physical world

As you can see from Figure 3.7, each layer is built from the things in the layer below. This concept is the same for the meta-model. The meta-model describes what can be built in the model (or the layer above). If you are able to update and change the meta-model, then you can create a model for the actual domain that you are working on. This is the second law, control the meta-model to fit the domain.

Let's move onto the third law, making models useful. How do you make models useful?

For a codebot, it uses the models to write code. Similar to the way humans take requirements, understand them, and then write code to fulfil the requirements. A codebot uses patterns in the target application and writes code to those patterns based on the model. To make a model useful like this, the model undergoes a series of transformations. Broadly speaking, there are Model-To-Model (M2M) and Model-To-Text (M2T) transformations that can be applied to a model. The model can either be text, diagram, or some combination of both to make a hybrid model. It is really up to the team involved in the project to decide what is the best approach for the model and the transformations to come up with a solution.

The third law is particularly interesting when it comes to legacy system migrations. A typical scenario when building a software application and the subsequent legacy migration occurs can be seen in Figure 1.1. In this project you can see that when it comes to the legacy migration project at a later point in time, the weak or lack of a connection between the model and the source code severely hampers the project.

A real world example of this is security update for applications. Imagine a security update has been posted and it effect many of the software applications in your ecosystem. If they were all connected using the models and transformations, the developer would only need to apply the security update to the transformations and then the codebots can write the update to as many applications as required.

It is beyond the scope of this book to go any deeper into the third law of making models useful. There are entire text books devoted to this [33, 32]. The important take away is that through the transformations, a codebot can write over 90% of the target application that a human would typically have to write. In summary, our three laws of modelling are:

- The first law states that models will always be incomplete.
- The second law states to control the meta-model to fit the domain.
- The third law states to make the models useful through transformations.

The first law recognises Bonini's Paradox, as a model of a complex system becomes more complete, it becomes less understandable. So, there will always be a gap between the model and reality. The second law is to control the meta-model. Once you have control of the meta-model, it is possible to create all sorts of cool models that fit the domain you are working on. And finally, the third law is about making the models useful. For too many years software engineers have created models that end up included in the graveyard of business cases inside word documents. The model has not been made useful and even worse, it becomes an overhead as it needs to be updated and manually kept in sync with any changes to the requirements. Eventually, the model is not updated and becomes out of date and ignored.

A key differentiator of modelling platforms is how they deal with these laws. Some platforms try to make the models complete and you end up using models which are more complex than just writing the code and harder to understand. Other platforms allow code from the target application to be added to the model, or they create a programming language so that more and more complex algorithms can be included in the model. Again, the models end up less understandable than actually just coding it.

The codebots approach is simple yet powerful in its approach to the laws. We do not expect the codebots to write the complex algorithms. We leave this up to the human. But we do not add it to the model to make it overly complex. The codebots write their code and commit it to a source repository and what they write looks like a human has written. This means humans can pull the code from the source repository and add their complex algorithms alongside the bot written code. We truly like this approach as it also considers the future of work and how we use technology. The bots do the bulk of the project around the heavy lifting. The humans do the complex algorithms and the creative parts of the projects.

3.6. Summary

In this chapter we have presented a strategy and five tactics to help you on your continuous modernisation journey. The strategy is the overarching plan. The tactics are specific actions undertaken as part of the strategy.

The strategy is to draw on the lessons of the past and use known military tactics to help you get underway on modernising your legacy systems. There is probably one particular legacy system that is holding back your organisation the most. But this legacy system is a monolith and it tightly integrated with a number of satellite legacy systems that surround the monolith. To eventually modernise the monolith, you will first need to modernise the surrounding satellite legacy systems. The strategy is to use a three phase Blitzrieg.

In this scenario, *Phase 1* is to modernise a satellite legacy system, *phase 2* is to map and connect to other systems, and *phase 3* is to modernise the next satellite legacy system until the monolith has been sufficiently isolated. For the smaller legacy systems you can use the firecracker migration pattern (see Section 2.3) and for the larger legacy systems like the monolith, you can use the divide-and-conquer migration pattern (see Section 2.4).

While undertaking this strategy we recommend five tactics that will enable you to gain momentum and reduce the costs associated with the effort required to modernise a legacy system.

The five tactics are:

1. Use science to drive innovation at a pace in step with the organisational rhythm.

2. Enable the community around your organisation. A strong community builds a defendable moat around your organisation.

3. Build small loosely coupled microservice applications emphasising a separation of concerns.

4. Increase the visibility and control of the entire software ecosystem by lifting the fog of war.

5. Use models to capture knowledge and enable control through the use of codebots to do the heavy lifting on software projects.

Moving the needle in the direction of any of these tactics will have an impact on the business agility of your organisation. It is our intention to continuously modernise the Codebots platform to help you achieve these tactics. That is quite a meta statement as we are continuously modernising our platform so that you can continuously modernise your organisation. We love everything meta! Applying our recommended strategy and tactics to ourselves is a great acid test. Some people call this eating your own dog food but we prefer the saying drinking your own champagne.

4. Codebots

“Codebots? That’s cheating!” Leo Mylonas, before he became Codebots Lead Engineer

Codebots are software robots that write code alongside your human team. On average, they write over 90% of the code base that a human usually has to write. In this chapter, we will unveil what a codebot is and answer many of the frequently asked questions we receive. Before we dive in too deep, let's start with an easy one about the name itself.

Codebots with a capital 'C' refers to the company name. You will also see us use a lowercase codebots as well. We like to think of codebots as a species like a fish. There are also specific types of codebots like Javabot, Csharpbot, and Lampbot, like there are specific types of fish like Giant Trevally, Black Marlin and Sail Fish. So, we may refer to the codebots like a species or sometimes use the word codebot as a general term to describe one of the codebots themselves.

The next general question we usually get is, what does a codebot look like? Besides some of the cool graphic designs our team has done, a codebot is pure software so it cannot be seen like a hardware device. However, software engineers usually draw architectural diagrams to depict software. A codebot is a software agent and uses architectures as described by some of the classic textbooks on AI [17, 29]. For example, one of the simplest agent architectures is the reactive architecture as seen in Figure 4.1. The reactive architecture uses agent behaviours that are a simple mapping between stimulus and response. The agent has no decision-making skills. We used a reactive architecture while experimenting on some of the early codebots before moving onto more advanced agent architectures.

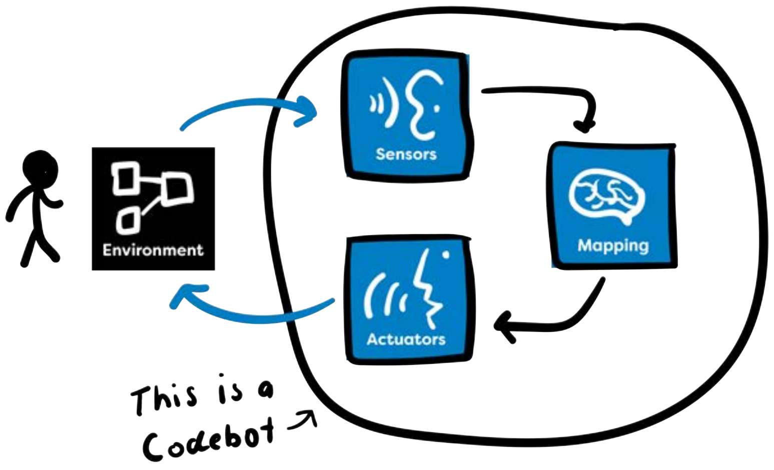


Figure 4.1: Reactive architecture

Even though the reactive architecture is quite simple, it does facilitate discussions about some very important points. What are the sensors or the input? What are the actuators or the output? And what happens in the mapping to allow a codebot to write code? Now this is where we start to demystify a codebot and start drawing analogies to how we behave as humans. Effectively, we want a codebot to be like another member of the team. When we work on software projects with all humans, we communicate our ideas and intentions of the software using many different approaches. No doubt, we will communicate using voice in a meeting. We would also use language in some email or in a requirements documents of some description. And finally, our favourite, we would have some super focused whiteboard sessions where we would draw some diagrams and other cool things to represent the software project.

These are all candidates for the sensors or the input to the codebot: diagrams, language, voice, and text. For example, through the platform chat, you can ask a codebot to deploy the application using something like ”@Lampbot, can you deploy the latest version?”. Lampbot would confirm the order and then report back once the application has been deployed. This text response back from the codebot is an example of an actuator, or some output. Another example of an actuator is the codebot writing some code. Once the codebot completes writing its code, it commits it up to a source code repository like Git[1]. If required, the humans can pull the source code from the repository and add to it as required. This leads into the next frequently asked question, what does the source code look like? And for the more technically savvy, how is this different from using a code generator?

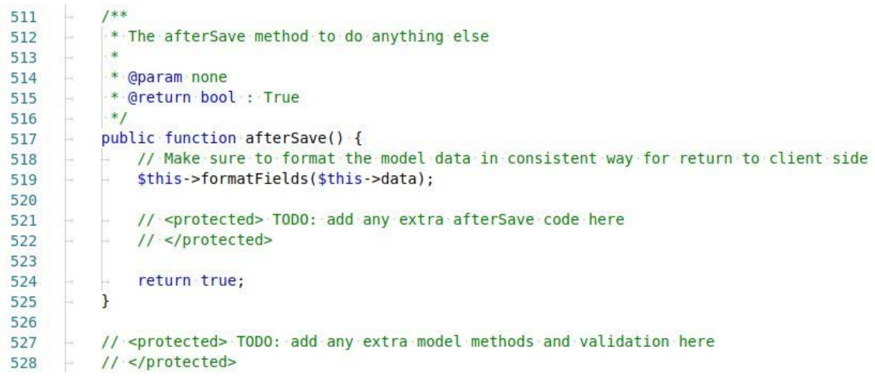
Code generators have had a long history in computing and can provoke some intense emotions in software engineers. One of the main reasons that software engineers end up disliking code generators is because what they produce can be a jumble of code that they (or any human for that matter) cannot understand. This creates a black-box and software engineers become frustrated as

[1]Git is a version-control system found at `https://git-scm.com/`

they cannot code the solution the way they would like, or understand what is inside the black-box and why their code is simply not working the way it should. As you could imagine, this has created a stigma around the use of code generators. So much so, that we have banned the word *generator* from use in our company. And if you do use it, you have to put one dollar into the swear jar.

To help leave the mindset surrounding code generators in the past, we have a simple rule that we follow; *a codebot must write code that looks like a human has written it.* This leads to many benefits. Firstly, the code written by a codebot becomes a white-box. Software engineers can lift up the hood of the engine and understand what is going on inside. This removes much of the frustration from the use of code generators but more importantly, makes the codebot a member of the team like the humans. So, when a codebot writes some code for a project and commits to the code repository, the humans can pull the code from the repository, understand it, and add some code alongside the codebot as they would do for any member of the team. And how is this achieved? Figure 4.2 is some code that a codebot has written. As you can see, it looks like a human has written it, however, if you look closer at lines 521 and 522, this is what is called a protected region. You can add code inside the protected region and the next time the codebots writes its code, your code will be preserved.

On average, a codebot can write about 92% of the code base. The other 8%, is written into protected regions. What is super



```
/**
 * The afterSave method to do anything else
 *
 * @param none
 * @return bool : True
 */
public function afterSave() {
    // Make sure to format the model data in consistent way for return to client side
    $this->formatFields($this->data);

    // <protected> TODO: add any extra afterSave code here
    // </protected>

    return true;
}

// <protected> TODO: add any extra model methods and validation here
// </protected>
```

Figure 4.2: Protected regions

important to understand is that it is not expected that a codebot will write the entire application for you. In some cases, it might be possible, but it is not a goal of ours to have our codebots write all the code. We want the humans to remain the masters of creativity and to work on the truly complex parts of the application. Whilst the codebots help with the heavy lifting of the application, when they cannot write the code we need or are simply getting in the way for some reason, we must have an escape clause so that we can push the codebot to the side, and deliver the application within the allocated time. This keeps the application as a white-box and we use codebots for parts of the application that they are good at. Think of it like any team member, some are good at certain things but not so good at others.

We have now discussed some of the possible sensors and actuators of the reactive architecture from Figure 4.1. For the sensors, we can use diagrams, language, voice, and text as input to a codebot. For the actuators, we can use the same, but the text coming out can be code. So, what about the mapping in the reactive architecture? The answer comes from some lesser-known areas of software engineering that we believe holds the key to breaking the cycle of legacy. The areas include Model-Driven Engineering (MDE), Domain-Specific Language (DSL), and Software Product Lines (SPL). There is some conceptual overlaps from these areas, which contains the use of models. So, a codebot uses a model as part of the mapping from sensors to actuators. The model is the internal representation of the application that a codebot understands. In Section 3.5, we dive deeper into the world of models and advantages they bring to a software project.

Before we dive into the diagrams used by a codebot, let's take a quick look at some of the real-world results we have been able to achieve so far. Back when the first author of this book was a PhD student and producing research papers[5, 6, 7], there was always a need to run experiments and collect data to support the theory. An ideal experiment would have two teams, one using a codebot and one using only traditional software development. The teams would both work on the same application, over the same period of time, and then compare the results. While this sounds like a great idea, securing the resources, budget, and teams of relatively equal experience was impossible for a lowly PhD student.

It was not until some years after graduating from the PhD that we were finally able to conduct some experiments like this as we had grown the company to a sufficient size that we could afford the resources, budget, and teams. The experiment we ran was to have a software team work on a problem that was presented by an organisation. The team would then work on the solution for 1 week (5 business days) and then present the results. The organisation was then asked to fill out a survey to compare the results for how long they believed it would have taken them internally to complete. The results were very promising with the survey respondents giving an average of 8.3 weeks for what we were able to achieve in a week. The results from the Codebots field trials and the survey can found in B.

4.1. State and Behaviour

Since the dawn of computing, software applications have generally be divided into state and behaviour. It is a good, broad approach to thinking about a software application. *The state of an application is the data and the behaviour is what we do with it*. The importance of data for modern businesses can not be over emphasised. Data is so many things, including the collective knowledge of the organisation. And when data is hidden away in legacy systems like old databases and spreadsheets, the ability for the organisation to access that knowledge to make informed decisions is severely hampered.

Over the next three sections we are going to look at the diagrams we generally use for a codebot. Each software application has one model which is made up of many diagrams[2]. You can think of the diagrams like views or windows into parts of the model. The diagrams are how we represent the state and behaviour of a software application. In Section 4.2 we look at the entity diagram, which is the classic data model, but there are ways to add behaviour that simplify the overall model. In Section 4.3 we look at the UI diagram and how you can use it to add in some behaviours. And finally in Section 4.4 we examine the security diagram and how you can lock down and secure your application.

Before we dive into the diagrams there is one last point on the diagrams we want to cover. For some software applications these diagrams are not the ones you will want to use. The three diagrams we are presenting next are good for some codebots, but not all. We have built codebots that use a completely different diagraming approach. This is important to acknowledge because much of the mantra that we discussed in Section 3.5 on modelling, was centred on the three laws with the ability to start small and create solutions for specific domains. So, following this thought process, there will be some domains that do not suit our standard three diagrams. We have built the Codebots platform to be able to support different diagrams in the future.

[2]There is one model for each application. A model can have one or more diagrams. This is our definition of the relationship between a model and a diagram.

4.2. Entity Diagram

This is the classic diagram that all new students to software are taught and it has gone by many names, including the class diagram or the entity-relationship diagram. Figure 4.3 shows the codebots version of this type of diagram and is something you will become very familiar with on the platform.

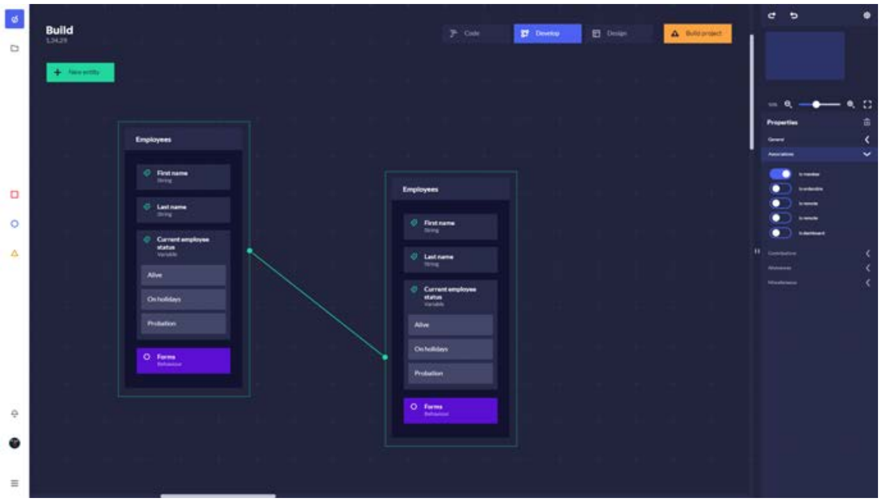


Figure 4.3: Entity diagram

Teaching you how to build a classic database schema is beyond the scope of this book and there are plenty of great resources available. If you have not read it, we recommend this book from 1978 [18] as is still relevant today. It illustrates the science and art of data modelling and contains a great discussion on the inherent problems and nature of information systems. So, we will assume that you are comfortable with building a database schema from here. The terminology we use throughout will be familiar and includes entities, attributes, validators, and relationships.

Our entity diagram goes a little further than the classic types and allows the modeller to connect the entities to parts of the UI and Security diagrams. For example, an entity may be part of a workflow and that entity would then move through the workflow in the target application. To achieve this, behaviours can be added to the entity as seen in Figure 4.4. The purple rectangle embedded under the attributes on the entity represents the behaviour.

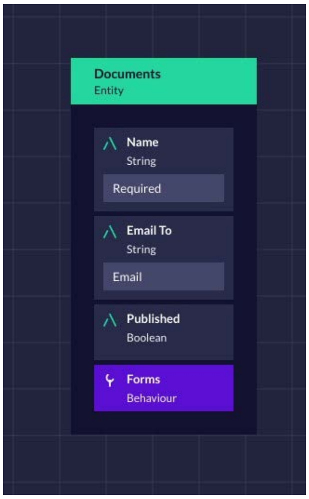


Figure 4.4: Configuring a behaviour with an entity

4.3. User Interface (UI) Diagram

This diagram represents the UI of the application and provides a way to show a high-level User Experience (UX) map of the application. To draw a comparison, you may be familiar with products like Adobe XD[3] or InVision[4]; these are hot-spotting frameworks where a designer can use graphics with hotspots to simulate what the final application may look like. It is our vision for the UI diagram to allow designers to rapidly prototype their designs like these other products, but to be able to have a codebot write the full-stack application instead of a click-through prototype. In this section we will be covering the various functionalities of the UI diagram and how it can be connected to entities and switch on behaviours to get advanced functionality.

Figure 4.5 is an example of a UI diagram. Once you have the diagram open, you have 5 different levels that you can work on. The levels are pages, tiles, views, components, and elements. The relationship between them is pretty straightforward to remember. A page can have many tiles, a tile can have many views, a view can have many components, and components can have many elements. Pages are the different URLs of the application and tiles are the various areas found on the page. For the developers, tiles are like microservices and how functionality is grouped and delivered on a page.

[3]Adobe XD can be found at `https://www.adobe.com/au/products/xd.html`
[4]InVision can be found at `https://www.invisionapp.com/`

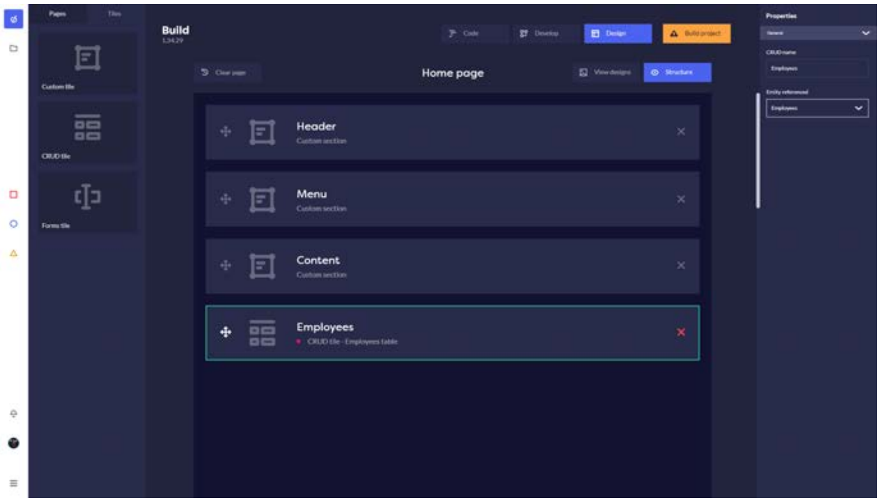


Figure 4.5: UI diagram

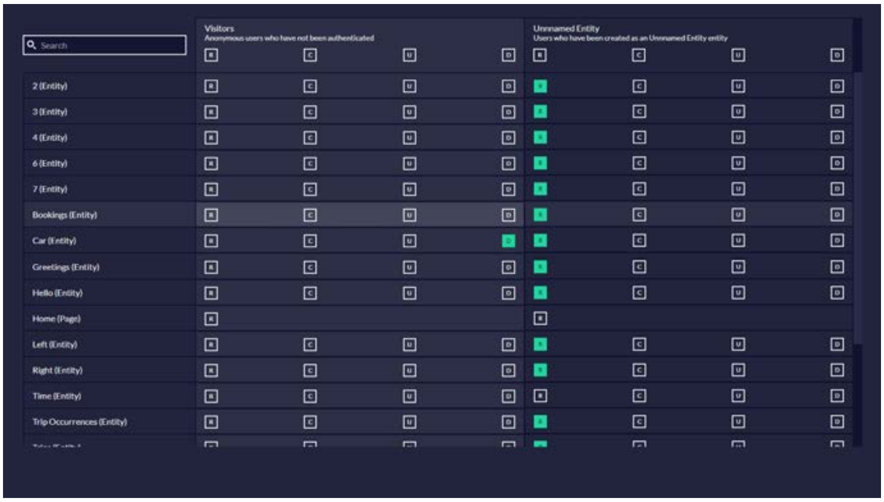
To use some of the behaviours of a codebot (see Appendix C for the list of behaviours) you drag a tile onto a page. For example, one of the most popular behaviours is the forms behaviour and it can be used for surveys and other ways to capture data. The modeller drags the forms tile onto the page and configures the related entity. This is all that needs to be done to then deliver this complex behaviour into the target application.

Further to these tiles, there are ways to customise the target application. There is a custom tile, view, component, and element. The modeller can add a custom object on any level and the codebot will put all of the code in place and then a developer can go to the source code and customise the application how they like. This is a really neat feature because it means that on any level or point in time you can create something custom to deliver a project.

4.4. Security Diagram

The security diagram is the last of the three diagrams and by no means the least important. Your ability to control access to the data and the application is paramount. The security diagram is split into three sections for Authentication, Authorisation, and Auditing (AAA). Authentication is how the user is verified. Authorisation is what tasks or data the user is allow to access. Auditing, sometimes referred to as Accounting, is keeping track of who does what so that it can be measured or forensics can take place if a suspected breach occurs.

In Figure 4.6, you can see the Authorisation matrix used for an application. On the left-hand side, or the rows, are a list of the entities. At the top, or the columns, are the user groups configured for the application. The modeller can then click in the matrix to turn the access on and off for the various user groups to that entity for the standard Create/Read/Update/Delete (CRUD) endpoints. For example, the modeller can grant access to the read API endpoint for a user group but not the ability to create, update, or delete. This level of control for the modeller gives them excellent control over the security of the application.



Search	Visitors Anonymous users who have not been authenticated				Unnamed Entity Users who have been created as an Unnamed Entity entity			
	R	C	U	D	R	C	U	D
2 (Entity)	R	C	U	D	[✓ R]	C	U	D
3 (Entity)	R	C	U	D	[✓ R]	C	U	D
4 (Entity)	R	C	U	D	[✓ R]	C	U	D
6 (Entity)	R	C	U	D	[✓ R]	C	U	D
7 (Entity)	R	C	U	D	[✓ R]	C	U	D
Bookings (Entity)	R	C	U	D	[✓ R]	C	U	D
Car (Entity)	R	C	U	[✓ D]	[✓ R]	C	U	D
Greetings (Entity)	R	C	U	D	[✓ R]	C	U	D
Hello (Entity)	R	C	U	D	[✓ R]	C	U	D
Home (Page)	R				R			
Left (Entity)	R	C	U	D	[✓ R]	C	U	D
Right (Entity)	R	C	U	D	[✓ R]	C	U	D
Time (Entity)	R	C	U	D	R	C	U	D
Trip Occurrences (Entity)	R	C	U	D	[✓ R]	C	U	D

Figure 4.6: Security diagram

4.5. Development Target

To be adopted by software engineers and designers, a codebot must provide a means to leverage the software patterns and architectural styles that are relevant to the organisation, instead of those proscribed by black-box tools without any ability to adapt. For example, an organisation may prefer to use the .Net framework coupled with the React JavaScript framework for the UI. This means that we must be able to create a codebot to support this development target. In this section, we will explain how a codebot can evolve to support new technologies and how this process unfolds.

For those unfamiliar with programming, software engineers use patterns in the application to create solutions. Patterns [12, 11, 2] are well-known and they have been written about for many decades. Some of the most well-used patterns such as Model View Controller (MVC) were first proposed in 1978 [26] and are still prevalent today. In danger of over simplifying, a software architecture is the combination and use of these patterns in the application. You can think of patterns like different types of lego blocks you can add together to make something. Most software engineers have a preference for certain patterns over others and there are entire conferences dedicated to learning new ways of building software. Some of the more recent architectures like microservices are discussed in Section 3.3.

In this section, we describe how to set out a process to build a codebot. While other organisations will not tell you their inner workings because they believe this is their intellectual property, we believe it is important that everyone in the community knows how to do this so you are armed with the knowledge on building your own codebots. This means that any organisation can satisfy their unique technology requirements. For example, we have a codebot called Javabot that uses the popular Spring and Hibernate frameworks, but if you did not want to use Spring and use J2EE instead, then you will have a path to follow to achieve this.

There are three flowchart diagrams coming up over the next few pages. These diagrams are the process we follow to create and update a codebot. Starting with Figure 4.8, there is a *Start Iteration* and *End Iteration* that represent the beginning and end of an Agile sprint. Recall that Codebots is complimentary to Agile and other software development methodologies and not a replacement. The first question in the flowchart is *Can the bot write the code required for the user story?* At the beginning of any new codebot, the answer will always be no. This means that we *write the code as a reference implementation* with the team implementing the user story just like they usually would. This simple first step of writing the reference implementation in this way has many far-reaching consequences that we will dive deeper into at the end of this section, but for the moment let's continue around the flow chart. The next step is to train the bot and check the code matches the reference implementation. Once it matches the references implementation, we build a model, get the bot to write the code, test the target application satisfies the requirement, and end the iteration.

It may seem counter-intuitive to build the reference implementation first, but this is done with the knowledge that many applications of this technology stack will be built. So how do you know when there is a return on investment for the effort it takes to build a codebot compared to just writing the application? There are some formula's you can use like the ones used in Software Product Lines [4], though in our experience we make productivity gains on building the first application in the family, let alone the many that will follow. It is also important to note that

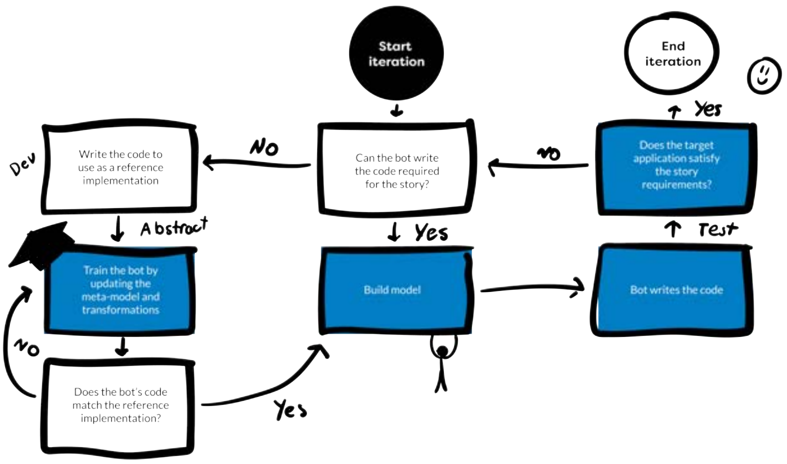

Start iteration
End iteration
Write the code to use as a reference implementation
Dev
No
Can the bot write the code required for the story?
No
Does the target application satisfy the story requirements?
Yes
Abstract
Yes
Test
Train the bot by updating the meta-model and transformations
Build model
Bot writes the code
No
Does the bot's code match the reference implementation?
Yes

Figure 4.7: Evolving a codebot

building your own codebot like this is only available through our premium product offerings and this is how we build a codebot specific to a customers unique technology requirements.

We often get asked the questions about the use of Machine Learning when we train the codebot. The answer at the moment is that we do not use Machine Learning (ML) for this yet. Using ML to solve this part of the overall problem is notoriously difficult. For example, some research coming out of University of Cambridge and Microsoft [1] has been able to use ML to solve problems of difficulty comparable to the simplest problems on programming competition websites. While this research is excellent steps forward, it is still a long way from building full-stack applications like a codebot. The codebots use AI in different parts of their internal technology stack as discussed at the beginning of this chapter. But will we be using ML for this in the future? We can't reveal all our secrets but we would suggest that the solution is not quite what you may think.

After a few iterations of evolving a codebot, some user stories will be presented that cover the path shown in Figure 4.7. This occurs when the answer is yes to *Can the bot write the code required for the user story?* What we have found is that there are reoccurring requirements that are found across different parts of the organisation. For example, it is common to want to survey or present some questions to end users to collect data of some description. This is so common that it is one of the first behaviours that we recommend training a codebot on. We call this the forms behaviour and we have rarely delivered a project without using it. The forms behaviour is also very handy for migrating from old PDFs and this is covered in Section 5.10 of the migration kit.

The third and final flowchart about evolving a codebot for a development target is shown in Figure 4.9 and this one is our favourite. You will notice that it is the superset of the other two flowcharts and includes an extra question if the bot cannot write the code for the use story; *Is it warranted to train the bot to support this in the future?* If the answer is yes, then you follow the path described above. If the answer is no, then the *code is written by a developer in a protected region*. This path is the other 8% on average that is not written by a codebot and has two very important consequences.

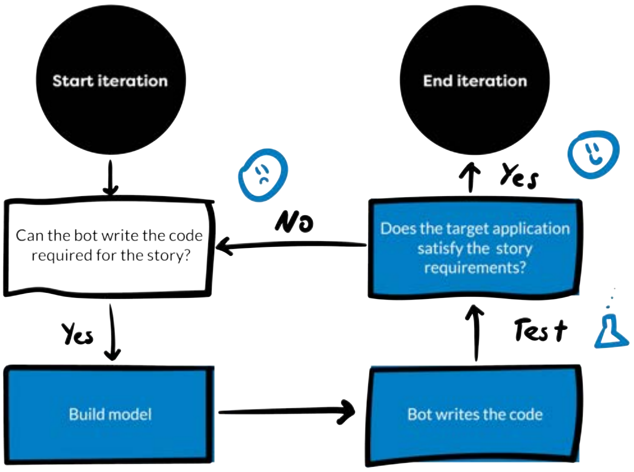

Start iteration
End iteration
Can the bot write the code required for the story?
No
Does the target application satisfy the story requirements?
Yes
Yes
Test
Build model
Bot writes the code

Figure 4.8: The bot can write all of the target application

Firstly, at any point in the software development lifecycle, if the codebot cannot do the work for you, you can push the codebot to the side and return to traditional software development. This means that using a codebot will always be faster because the base case is that you build the software like you normally would. Secondly, we love the implications this has on the future of work. A codebot is not here to replace us but to do the heavy lifting, leaving us to the truly complex and creative parts of the solution.

You will have noticed that at no point in this section we have discussed or shown a single line of code and that is for good reason. It is inconsequential what the development target is. As long as the development target uses patterns, which all software applications

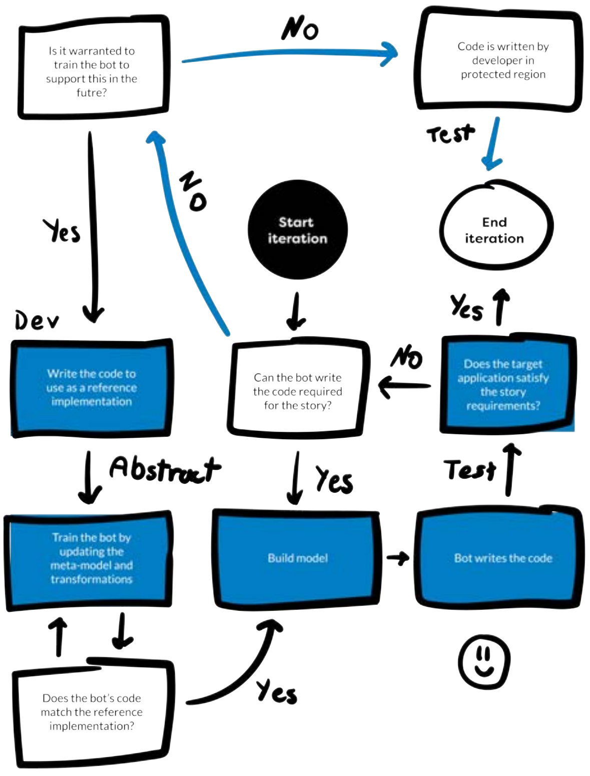

Is it warranted to train the bot to support this in the futre?
No
Code is written by developer in protected region
Test
Yes
No
Start iteration
End iteration
Dev
Yes
Write the code to use as a reference implementation
Can the bot write the code required for the story?
No
Does the target application satisfy the story requirements?
Abstract
Yes
Test
Train the bot by updating the meta-model and transformations
Build model
Bot writes the code
Does the bot's code match the reference implementation?
Yes

Figure 4.9: A human adds in extra code to finish the requirement

do, then a codebot can be built for that development target. At the heart of what we are doing here is making time an ally of the organisation, and not an enemy. Software teams are always on a never-ending deadline from one iteration to the next and this can have some unintended consequences. People become rushed because they feel they do not have enough time. But we can make time an ally of the organisation by evolving a codebot naturally alongside the software teams. As time progresses forward, and it inevitably will, your codebot will continue to improve if you follow the process. This is what we have observed and there are a number of other benefits that we will discuss now to round out this section.

Imagine yourself in the future where your software team has gone through several iterations and the codebot has evolved some pretty cool behaviours. You have made sure that the security is tight and paid attention to your Authentication, Authorisation, and Auditing (AAA). Authorisation has been reinforced with two-factor, authorisation includes security schemes for tighter control, and auditing provides a log of activity to enable forensics on user activity. You are feeling pretty good that you have the security bases covered, but an alert is sent out from a vendor with a security vulnerability. Traditionally, you would need to check all of the different applications across the eco-system to see if they are vulnerable, and if they are, update each application individually to fix the vulnerability. But in this future state you have a codebot on your team. You would only need to make the changes to a single reference implementation and train the codebot once. The codebot could then write to as many applications as required to fix the vulnerability; it could be hundreds of applications. By using a codebot you have a consistent architecture and you can deal with wide-reaching problems such as security in a more timely manner. This is only possible because you control the development target and are masters of your own destiny and not at the mercy of a third-party vendor.

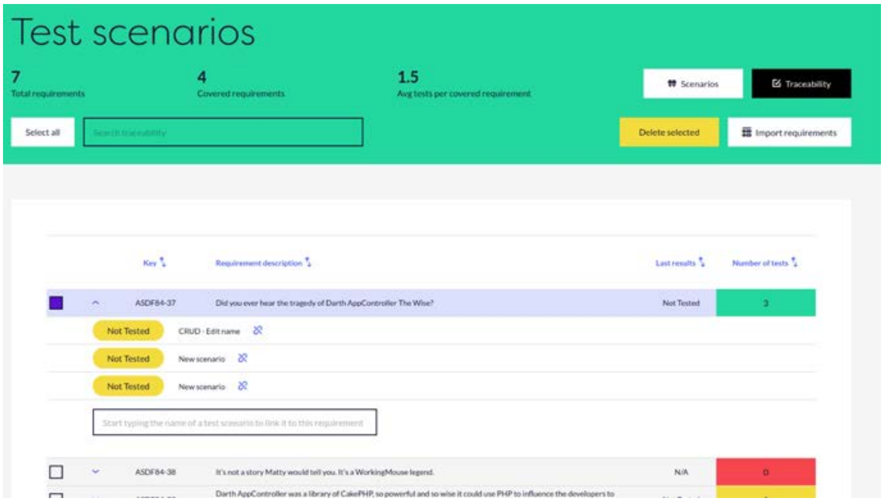
Another benefit worth addressing around consistency is how the codebots write code using patterns in the development target. Consistent use of patterns across a code base can go a long way to reducing technical debt. What happens over the life of a project is different software engineers come and go from a project and they bring in their preference to how patterns should be used to solve problems. Usually, and unwittingly, the different implementations of the patterns end up being in conflict and the developers back themselves into a corner. Have you ever been told by a software team that there is a delay because of a technical issue? Well, this could be one of the reasons: that the team is not able to get the architecture to work because of some conflicts in the patterns they are using. We must make the consistent use of patterns an important practice to minimise technical debt. From an architectural viewpoint, a codebot is an opportunity to ensure the consistent use of patterns across the code base and this has a positive flow-on effect to the maintainability of the application.

4.6. Testing Target

The development target is only half of the picture; the other half is the testing target. The use of codebots can extend much further than just development and in this section we are going to look at the benefits with testing. Think of it like owning a Ferrari; do you want to drive around just in first gear? We would say not. The good news is that a codebot writes the testing target as well. Before we get into the specifics of this, let's take a broader look at the role of testing in a software project.

The ultimate insight for any stakeholder in a software project is knowing that the software quality is high and you can sleep soundly at night. To achieve this, there is a simple yet powerful approach that can provide this assurance. The answer is to use a traceability matrix. A *testing and requirements traceability matrix* is shown in Figure 4.10. The columns are the requirements and the rows are the tests. This makes it possible to specify which tests relate to which requirements.

This traceability matrix leads into all sorts of great metrics about the project and helps answer important questions. For example, how many requirements from the backlog do not have tests before a release? Or even more simple, how many tests are currently passing? Knowing the answers to these questions can significantly reduce the risk of releasing poor quality software. Testing is a key contributor to quality and inadequate testing can


Test scenarios

7
Total requirements

4
Covered requirements

1.5
Avg tests per covered requirement

Key	Requirement description	Last results	Number of tests
ASDF84-37	Did you ever hear the tragedy of Darth AppController The Wise?	Not Tested	3
Not Tested	CRUD - Edit name		
Not Tested	New scenario		
Not Tested	New scenario		
ASDF84-38	It's not a story Matty would tell you. It's a WorkingMouse legend.	N/A	0
	Darth AppController was a library of CakePHP, so powerful and so wise it could use PHP to influence the developers to		

Start typing the name of a test scenario to link it to this requirement

Figure 4.10: Testing and requirements traceability matrix

leave an organisation exposed to unnecessary risk. Testing is used to help confirm whether a System Under Test (SUT) is behaving as expected and a traceability matrix goes a long way to support this.

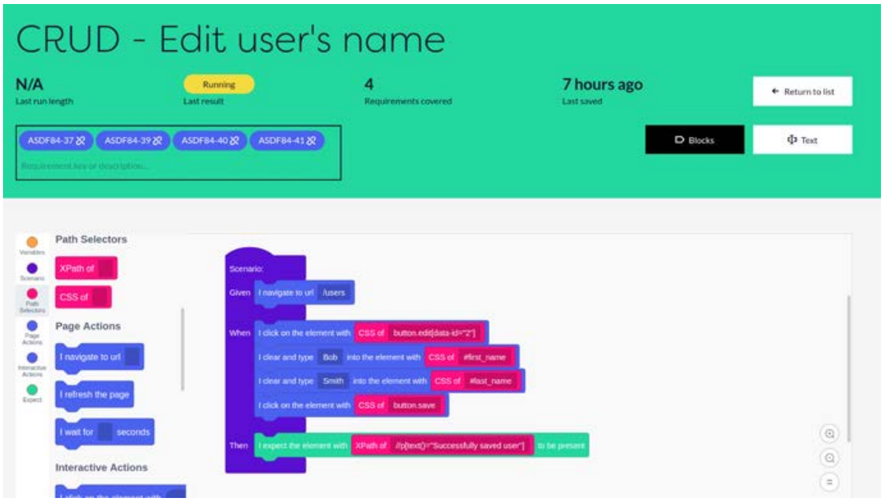
There are many challenges with testing that make it difficult to manage. For any non-trivially sized system, it is impossible to have a SUT 100% tested. The state explosion problem means that there are too many combinations to test and current computing power means that you would be waiting a very long time to test them all. So, testing of a system must be declared adequate for an organisation and this is what makes it difficult to manage; i.e. what is considered adequate? There is no such thing as fully tested; it is a myth.

This makes testing a sliding scale. Smaller organisations that produce simple web applications for the mass market can move the testing slider to the low end of coverage (if any). If faults or bugs make it through to production, the consequences are minor. On the other hand, larger organisations that produce critical applications must move the testing slider to the higher (more rigorous) end of coverage as the consequences of errors are severe. On a traceability matrix, you can start looking at the density of tests around requirements and also introduce code coverage tools to gain further insight.

The best approach to testing is a layered approach similar to the way security is implemented. Having one layer of security is risky and it's better to have several layers of defence. If one layer is compromised then there are still more layers before a breach occurs. A more secure system is subsequently achieved. A similarly layered approach is best for testing. In general, the 4 layers of testing are unit, integration, system and User Acceptance Testing (UAT). The better test systems use relationships and Continuous Integration (CI) to minimise the number of manual tasks.

Before we look at some of the techniques we use for a codebot to write tests, we want to draw your attention back to how the development target is built as described in Section 4.5. The testing target follows the same process alongside the development target; i.e. we get the codebots to write their own tests to ensure the development target works as expected. In addition, we use the same model for testing as we do for development. Theorists argue that this scenario runs into a *lack of independence*, but in our experience using the 4 layers of testing is more than enough to mitigate this risk.

Further to the codebots writing tests, it is possible to add in extra tests that cover more scenarios. Recall that the codebots write some but not all of the target application. So, for the extra code that is added by the human software engineer, you can add in extra tests as can be seen in Figure 4.11. We use scratch, a programming language used to teach children to code, which means adding tests just got a whole lot easier.



CRUD - Edit user's name

N/A Last run length · Running Last result · 4 Requirements covered · 7 hours ago Last saved

ASDF84-37 ASDF84-39 ASDF84-40 ASDF84-41

```
Scenario:
Given  I navigate to url /users
When   I click on the element with CSS of button.edit[data-id="2"]
       I clear and type Bob into the element with CSS of #first_name
       I clear and type Smith into the element with CSS of #last_name
       I click on the element with CSS of button.save
Then   I expect the element with XPath of //p[text()="Successfully saved user"] to be present
```

Figure 4.11: Build a test using scratch

For the rest of this section we are going to get a little more technical, for the non-techie readers you can skip forward to the chapter summary if you like. The rest of the section is going to look at how the codebots write their tests. Like the development target, we use a subset of MDE but from a research area called Model-Based Testing (MBT).

MBT differs from classical testing techniques as it uses a model to drive test generation. The potential benefits of MBT include higher fault detection, reduced testing cost and time, improved test quality, requirements defect detection, traceability and requirements evolution. The process is shown in Figure 4.12. The test case generator uses the model to generate test cases. The test script generator uses the test cases to generate test scripts. An adapter can be used to concretise the tests scripts and a test execution tool is used to execute the tests against the SUT.

If you are interested in this area, it is recommended that you read the book from Utting and Legeard [32]. We use a slight variation of the process shown in Figure 4.12, but it is generally very close. We have been able to apply the process to a number of different test frameworks such as Cucumber (`https://cucumber.io`) and SpecFlow (`https://specflow.org`). Importantly, the testing evolves alongside the development target as described during the last section in Figure 4.9, but now we will draw your attention to the question; *does the target application satisfy the story requirements?* The testing target must be used to answer this. Green means yes!

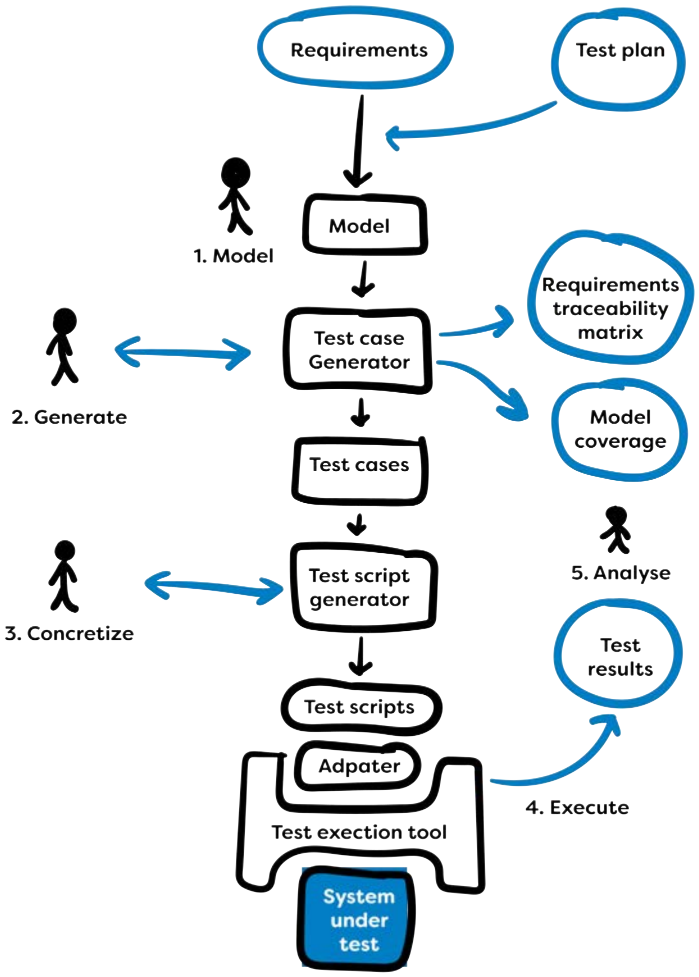


Figure 4.12: Model-based testing process adapted from [32]

4.7. Summary

Codebots offer a number of benefits for an organisation. They deliver quality software at speed and allow reuse at scale. The Codebots field trials show teams working with a codebot 8.3 times faster than teams without. Furthermore, the behaviours can be reused many times with updates and improvements able to be rolled out at scale across multiple applications. These benefits alone make for a good business case and a return on investment by reducing the total cost of ownership for a program of work.

Many organisations are struggling to find skilled software teams as there is a world wide shortage. Some organisations look at moving their projects off-shore but this raises a whole set of extra challenges. Ideally, people want to work with an internal or local team that are in their timezone and speak their language. Bringing a codebot into your team gives you the best tools in the shed. This means that you can do more with your current resources as the codebot will do a lot of the heavy lifting for a project. Or, you can look at reducing the amount of resources needed for a project and redistribute them onto other work.

A codebot writes to a target application and we split this into the development target and the testing target. The development target can be any technology stack and we have built a codebot for many different programming languages and frameworks. As long as there are patterns in the target application, then a codebot can use those patterns to write vast amounts of the target source code. Codebots are best at projects where there is a large amount of source code where patterns repeat. For example, web-based systems and other large code bases are good candidates. If the code base is very customised for that particular scenario, then a codebot may not be suitable to use. Some examples of this are embedded systems and small projects with very focused goals.

Our codebots are not going to be turning into Skynet from the Terminator. You'd be surprised how many times this comes up when people first think about bots that code. The reason that we are confident with this statement is that we have a simple rule; *a codebot is only allowed to write a target application and it is not allowed to write its own code*. This rule circumvents future issues around any sort of runaway intelligence caused by the codebot updating its own code. This might sound like science fiction, but there are already movements across the globe about using AI responsibly and we want to make sure we do too.

The future development for the codebots is exciting. We are going to continue to expand out their sensors and actuators making them more like any member of the team. We are also moving forward into a position where you can create your own codebot. At the moment, you can either use one of the existing codebots like Springbot or Angularbot, or you can ask us to build a new codebot for you. In the future, we want you to be able to create and train your own codebot. This would allow a far better match for your technology stack and it would also mean you could create your own new behaviours. Have a look in Appnedix C for some of the behaviours we have built so far. But I am sure that the people in our community will come up with ones that we haven’t even thought of yet.

5. Migration Kit

The migration kit is a set of activities that can be used to continuously modernise your organisation. There are four stages that span both the scope and development stages of the project lifecycle and this is shown in Figure 5.1.

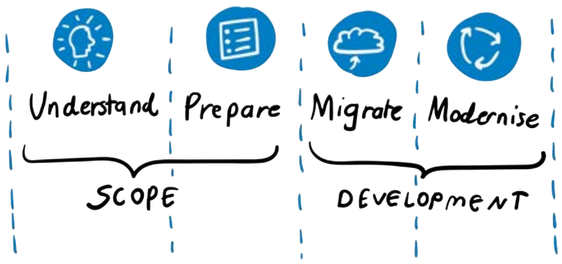


Figure 5.1: Four stages of the migration kit

The first stage is to *understand* the project. In this stage the team and the stakeholders gather the knowledge, reverse engineer requirements, and experiments to better understand the project. The second stage is to *prepare* the project. In this stage the layers of the community and decision making is mapped, expectations are managed, and stories are estimated. The third stage is to *Migrate* the project. In this stage the migration activities are described for databases, spreadsheets, and PDF forms. Finally, the *Modernise* stage is used to prepare for success in the future.

5.1. How to Use this Kit

Summary

A summary of the activity and why it is being applied. The how is covered in the Steps.

Level of difficulty

There are three levels of difficulty: *Easy*, *Moderate* and *Hard*. *Easy* activities don't always mean quick activities, but they are ones that should be quick to pick up and add value to a project. *Moderate* ones tend to need some pre-planning while *Hard* normally needs everyone to be on point for the activity. The fewer interruptions the better.

Before you Start

Each activity normally has a few requirements that need to be satisfied before you can start. Normally other activities flow into ones later in the process.

Stage

There are four stages that an activity will be categorised in: *Understand*, *Prepare*, *Migrate* and *Modernise*. The *Understand* and *Prepare* activities happen during scope. *Migrate* and *Modernise* activities happen during development. There is some overlap of activities and you can do the stages in staggered formation if needed.

Suggested time

This is how long the activity should take.

Participants

The participants of each stage vary depending on who is needed for each task. They include:

Product Owner - The main contact for the project.

Business Analyst - The domain expert that understands both the business and how to integrate with technology.

Squad lead - The scrum master of the project.

UX Designer - All designers on the project.

Web Engineer - The software engineer developing the target application.

Tools Engineer - The software developer testing the target application and improving tooling.

Account manager - The account manager for the project.

Stakeholders - Any members of the team that has a say in the product.

Users - Target user groups who are from the target demographic.

Materials

These are handy materials to be used for this activity.

Steps

Step-by-step instructions to follow for the activity. Some of the activities will have detailed steps and others will be more of a guide.

Justification

The justification contains any background, knowledge and references to further reinforce why this activity is important. The justification could be quite long or short depending on the activity.

5.2. Reverse Engineering Requirements

Summary

The stories backlog is a list of requirements detailing the functionality of an application. Creating and maintaining the stories backlog is one of the most important tasks that a team does throughout the software lifecycle. It provides everyone involved in the project a list of what is required, how important each requirement is, what users are involved in it, and what platform it belongs on. The backlog is the basis for how the project is estimated and the roadmap constructed. It is broken down into epics and user stories. The epics are large groupings of functionality while the stories are the small individual tasks the user performs on the product.

Level of difficulty

Moderate

Before you Start

This is one of the most important migration kit activities as it sets the stage for success. This activity is traditionally performed by a business analyst. Find as many domain experts on the legacy system as possible and any documentation that might give clues to the intention of the application. If the domain expert cannot be found, then someone must become the domain expert. This can be very much a computer forensics activity, so get your detective hat on.

Stage

Understand

Suggested time

Unknown, hopefully less than the maximum time allowed for an iteration (usually 2 weeks).

Participants

Business Analyst , Squad lead, and anyone else with knowledge of the legacy system.

Materials

Codebots

Jira or any other issue tracking software that you use (optional)

Steps

1

Decide on the location where the backlog of requirements will be kept. The backlog should become the single source of truth for the requirements of the application. For some organisations, they may already have a policy on this like inside Jira [1]. Alternatively, you can use the Codebots platform as we provide a planning section as shown in Figure 5.2. Codebots can be used in isolation or in combination with issue tracking software such as Jira.

2

Create a new project in Codebots and fill out the basics of the product, or select a previously migrated project.

[1] `https://www.atlassian.com/software/jira`. We love you Jira.

3

Gather all your research and idea development along with assumptions and start turning them into epics. These epics shouldn't be too general and sometimes large features should be broken down further.

An epic is a description of a coarse-grained requirement in a story form. It may represent a theme, which can be used to categorise a bunch of related User Stories. epics are created and maintained within the Codebots project. An epic may look something like:

As a *[type of user]* I want to *[do some action]* so that *[reason for action]*

4

Once you have the epics in place, start breaking them down into the stories. A user story is a description of a fine-grained requirement. Each user story belongs to exactly one epic and describes part of the epic in more detail. A user story is the unit of delivery for iterations, so by definition a user story must be sufficiently small to deliver in a single iteration. A user story may look something like:

As a *[type of user]* like *[persona]* at *[environment]*. I want to *[do something]* using *[device]* so that *[reason for task]*. This will [user goal].

5

Before the project is able to go to development, the stories backlog needs to be finalised and all information added. The more the better, as it keeps accountability of the project moving ahead.

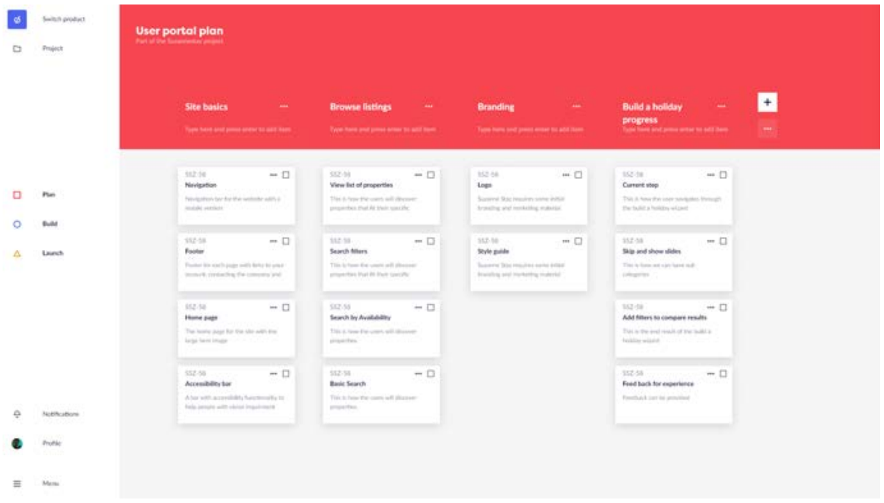


Figure 5.2: Plan UI

Justification

The role of a great business analyst combines both the art and science of understanding a domain. It is beyond the scope of this book to delve into the depths of this world as many have before us. In the academic world this is sometimes referred to as requirements elicitation and in the agile world as requirements gathering. As you can get a feel for [19, 13], this is a very widely discussed and written about topic.

Without over-simplifying this field, there are a few main points we would like to address. First and foremost, keep it simple. It is easy to get carried away with the complexity of requirements gathering and try to do too much too early. Modern business analysts are embracing the ethos of methodologies such as Lean thinking, where it is the MVP that is the first milestone to be achieved. That is a good way to think about it, however it needs to be applied in the context of legacy systems i.e., what is the minimum set of requirements needed to satisfy the user stories for the people using the legacy system. By setting the priorities of the user stories (requirements) and engaging with the people on the project early, you can carve out the MVP of the legacy migration project. In Chapter 2, we covered a few scenarios on how to deal with legacy systems. We looked at the firecracker migration pattern (see Figure 2.3) and the divide-and-conquer migration pattern (see Figure 2.4). Both of these should fire up your imagination on how to tackle the legacy system.

It is also worth revisiting some of the tactics we laid out in Chapter 3, especially the use of models. The insanity of legacy systems is that the cycle keeps repeating itself. When we rewrite an application or deploy a new off-the-shelf application we expect a different result, but we are simply creating the new legacy system of tomorrow and the cycle repeats. However, if we are able to make the requirements (the knowledge) of the application as important as the application itself, the next time we come to the point that we need to change the application to better fit the people using it, then we are already on the front foot because the whole idea of reverse engineering the requirements is a mute point. You will already have them.

5.3. Experimental Framework

Summary

There are many different testing frameworks and cool tools that are readily available. It is not our intention to reinvent the wheel and we encourage you to explore and choose a system that works for your organisation. The steps in this activity will guide you in planning, performing, measuring and acting on an experiment The important point is not which framework you choose, but that you embrace an experimental mindset and encourage an investigative culture within your team to help optimise business processes and projects.

Level of difficulty

Moderate

Before you Start

One of the best times to identify problems and have ideas is when you are talking to team members about the challenges that they are facing. Collecting both qualitative and quantitative data that supports the problem you want to solve is useful when pitching an experiment to other stakeholders.

Stage

Can be applied to all stages

Suggested time

The activity of setting up an experiment depends on the complexity and number of people involved but often it will take a couple of hours to a couple of days, depending on the knowledge gathering required.

Participants

All stakeholders and users can (and should) be involved in experimentation but not all at once! Keep your experiments limited to the participants who can add the most value.

Materials

Whiteboard.

Steps

1

Understand the problem. We call this phase of the process, discovery. It's time to observe, ask lots of questions, i.e. the 5 Whys [31] and dig into the historical data that you already have like burn-down charts or platform analytics. Interview the people closest to the problem, quiz them on their current processes and carefully map out their user journey so that you're ready for the next step.

2

Develop a hypothesis. Once you have analysed the problem, you must identify a solution. This is the really fun part as it involves brainstorming ideas for your potential hypothesis statement. A hypothesis is a prediction that you create prior to running an experiment. A good hypothesis makes it clear what it is you are validating or invalidating. A common format is: If [cause], then [effect], because [rationale]. Remember that there are no bad ideas.

Focus on nurturing a collaborative environment and looking at the problem from all angles based on the knowledge that you collected in the first step.

3

Plan the experiment. Now that the hypothesis is ready to test, you need to identify the most appropriate solution for the problem and create a strategy to test it. We call this the Implementation Plan and it requires that you decide: who is in charge; who is involved; the duration; identify the metrics to track/measure the experiment; and how the experiment will be implemented.

4

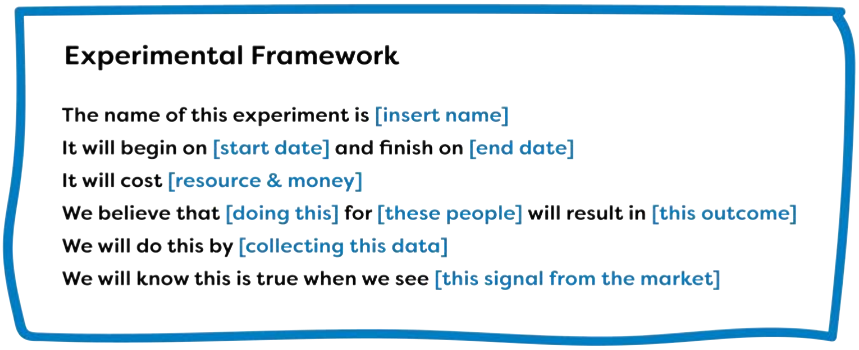
Collect the data. The main difference between a well-formulated hypothesis and a guess is, data. In the previous step you defined how to measure the experiment. It could be measured in money, time, community satisfaction, or another critical metric. Whatever you choose, now is the time to collect the data that defines a clear criteria for success and failure. And remember, failure should not be feared; it's all part of the learning.

5

Make a decision. At the end of the experiment you will analyse and interpret the data to determine if the experiment was successful. You can use the results to create new hypotheses or pivot the experiment to explore new solutions. Just make sure the whole process is documented and available for others in the company to view.

Justification

It's probably no surprise, given that one of our core values is *Scientific but not heartless*, that we are big fans of the scientific method, a problem-solving approach at the core of all sciences. We have also been influenced by the Lean Startup which is about running a lot of small experiments with a focus on metrics. A common misinterpretation of the Lean Startup's *build, measure, learn* feedback loop, is that you should start with *build*. We always start with *learn* as it encourages the team to think through and understand what they're trying to learn before they start building. That way you're building to learn, not just building to build. In Figure 5.3, we have included an example of a simple framework as a good starting point for your experiments.



Experimental Framework

The name of this experiment is [insert name]
It will begin on [start date] and finish on [end date]
It will cost [resource & money]
We believe that [doing this] for [these people] will result in [this outcome]
We will do this by [collecting this data]
We will know this is true when we see [this signal from the market]

Figure 5.3: A simple framework to get started

5.4. Community Onion

Summary

The purpose of this activity is to align on purpose and assign layers of responsibility as illustrated in Figure 5.4.

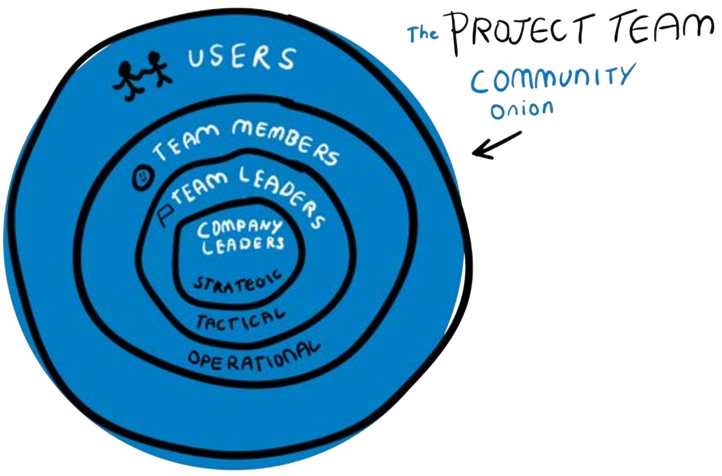


Figure 5.4: Project team community onion

Compared to Figure 3.4, the project team community onion is focused on the four levels of project participation, determined by whether a team member is making strategic, tactical, operational or user decisions. The activity ensures that those at the core of the onion who are most invested in making the change, are aligned on the mission and the vision of the project. Once this is established, it's purpose is to empower the project leaders and influencers to plan how the project will take shape. When these key stakeholders understand and believe in the mission and have influence over the project scope and resources, they are more willing to commit to a product roadmap. Essentially they become project ambassadors for the operational team members, who are responsible for what actually gets built. With full visibility as to why it is happening in the first place and how it's going to happen, the main project team is empowered to create exactly what the end-user requires and rally their support for the final solution. Two-way communication between end-user and all levels of decision making is important throughout the project to create the technical and cultural shift that ultimately results in finding software/people fit.

Level of difficulty

Easy to start, however ongoing effort is required to realise positive, long-term benefits.

Before you Start

The person who is responsible for instigating the project needs to the mission, the vision and the values that will underpin the project as it currently stands and bring these to the first session.

Stage

Prepare

Stage

Plan

Suggested time

The duration of this activity will be around two hours, plus regular ceremonies for each layer to ensure alignment and delivery of the project.

Participants

Management and top level stakeholders participate in the first step of the activity. Every person, from planning and development to end user is considered a member of the community and a potential participant in latter steps of this activity.

Materials

Whiteboard, an org chart, open minds and commitment to the mission.

Steps

1

Bring together the inner core of the community onion, the strategic decision makers of the project, i.e. CEO, CTO or CIO. Gain consensus on the vision and mission of the project by focusing on the *WHY*.

2

Take the project leaders on the journey by bringing together the next layers of the onion, the tactical decision makers, i.e. Product Owner, Business Analyst and Squad Lead. Share the vision and mission of the project and empower the leaders and influencers to decide and commit to the scope and the resources required for a successful project.

3

Now it’s time to bring the wider project team on the journey by involving the operational decision makers, i.e. Engineers, Designers, and Account managers. Share the project mission, vision, resources and scope and invite them to contribute to the form and function of the project by prioritising what needs to happen to achieve the project goals within the agreed timeframe.

4

Step 4 happens concurrently with steps 1, 2 and 3 and is critical to ensure internal adoption of the project. It involves two-way communication with the end-users of the product, sharing the vision, mission, scope and function of the project as well as understanding the cultural context, motivations and processes that currently exist.

Justification

This activity is a combination of three different concepts applied specifically to a software migration project. The Community Onion is a conceptual framework to think about the different kinds of membership a community can have. It is based on the idea that those at the centre have the highest level of commitment and contribution to the community.

Modern military theory divides war into strategic, operational, and tactical levels and we use these same definitions to define decisions made at different levels in an organisation’s hierarchy. In this book we use a lot of military tactics as they can easily be applied to an organisation in the pursuit of strategic goals. Strategic decisions are long-term in their impact and they shape the direction of the whole business. For example, the priority and timing of when a particular migration project should happen. They are generally made by senior executives. Tactical decisions help to implement the strategy and are usually made by project leaders or middle management. Operational decisions that relate to the day-to-day running of the project and are mainly made by team leaders or members.

5.5. Managing Expectations

Summary

Managing the expectations of the project length is extremely important to ensure project success. When the expectations are managed and highlighted early in a project, the team is able to work creatively without the additional pressure of overhanging deadlines and late deliveries. The trade-off sliders are a way to tease out of the team what is important to them and the cone of uncertainty helps ground the discussion. This activity can be used both for internal projects so management can be well informed, or used for external projects so that service providers and customers are in alignment.

Level of difficulty

Easy

Before you Start

Spend some time searching online and becoming familiar with trade-off sliders, the cone of uncertainty, and agile vs waterfall. There are plenty of articles that will give you some background.

Stage

Understand

Suggested time

30 - 60 mins.

Participants

Product Owner, Squad Lead, Account Manager, and any other

Stakeholders interested in or influencing the length of time for the project.

Materials

Whiteboard and markers.

Steps

1

On the whiteboard, draw up the cone of uncertainty. The cone shows that the further into the future that we try to predict, the less accurate we will become with our prediction. The justification in this section has more of an explanation on this. Spend a total of 10 mins asking each stakeholder why the cone of uncertainty is a practical reality. If you are struggling to get the stakeholders to agree, use a version of the Johari Window[2] and emphasise how the unknown unknowns compound over time to create a cone.

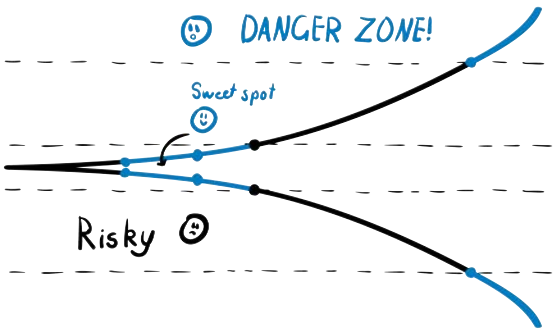


Figure 5.5: Cone of uncertainty

[2]Read about the Johari Window at `https://en.wikipedia.org/wiki/Johari_window`

2

On the whiteboard, draw up the trade-off sliders table and explain how only one choice per row and per column is allowed. Let the stakeholders in the room choose their locations in the table. Noting that they are not allowed to have chosen the same column twice, but they will try.

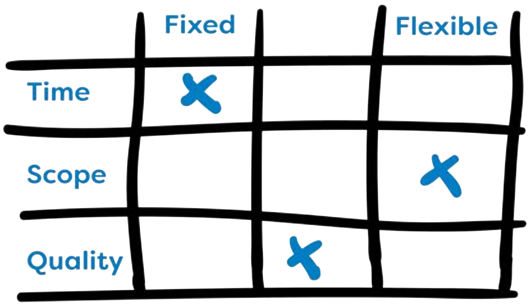


Figure 5.6: Trade-off sliders

3

If the stakeholder chooses time as more important over scope, draw a line on the time axis of the cone of uncertainty and explain how the amount of scope in this timeframe will be variable. This is fixed time, variable scope, and you can use an agile methodology for the project. This is the ideal scenario as it constrains the amount of time and therefore we know the costs but we do not know the scope. For internal projects, having fixed time can be useful for when deadlines are imposed on software teams. For external projects there must be a high-level of trust between the service provider and the organisation as the contract will be based on time and materials (T&M).

4

If the stakeholder chooses scope as more important over time, draw a line on the cone above the time axis and explain how the amount of time for this scope will be variable. For example, plus or minus 60%. This is variable time, fixed scope, and you can use an agile methodology. For internal projects, if the project finishes early and comes in under the cone i.e., less than plus 60%, then your manager will be pleasantly surprised. For external projects, ideally the contract will be based on T&M so that if the project is finished early, the customer will pay less. However, if the customer or the manager insists on a fixed price and not T&M, then proceed to the next step.

5

If the stakeholder is adamant that both scope and time are as important as each other, then the project is fixed time, fixed scope, and you should use a waterfall methodology. This is not an agile project. If they want to use agile start again at step 2. However, they may be happy to do a waterfall project. It is possible to give a maximum time using a formula based on the cone of uncertainty. For external projects, this will be a fixed price contract and you will need to use some old school tactics as scope creep becomes a significant risk. We have seen some service providers still do fixed priced contracts as agile, but they are very careful to use change requests for everything that is remotely outside of the original scope. The management of this is quiet time consuming and can be self defeating for all parties involved.

Justification

Zeno's paradox is a philosophical problem that is puzzling at its core. If you had to walk from point A to point B, we could halve the distance and you could move to the halfway point. Then from this new location, we could halve the distance again to point B and move this distance. We could halve this again and again and infinitely reduce the distance by a half. This implies it would take us infinite time to get from point A to point B because we can always halve the distance, yet we are still walking around without any problem! This is a paradox.

Estimating is a paradox too. We want to predict the future and be accurate, but we cannot be accurate because it is an estimate. This makes estimating puzzling at its core too.

In Physics, there is a phenomenon called the Heisenberg's uncertainty principle. The uncertainty principle states that the more precisely the position of some particle is determined, the less precisely its momentum can be known, and vice versa. In other words, you cannot know both its position and momentum; there is a trade-off.

Estimating has a trade-off too. Instead of position and momentum like the uncertainty principle, the trade-off is between scope and time. You cannot know both.

A scope is a plan describing the application you wish to build. In a way, a scope is a model of the application to be built. And like all models, they are inherently inaccurate. A first reaction is to spend more time scoping and make it more accurate, but a line is crossed during this process where you are no longer scoping and start developing. To truly know the scope you have to deliver the project to discover all of the inaccuracies, which defeats the purpose of a scope. In reality, a scope can only go so far within a reasonable time and will remain inaccurate.

Estimating the time for a project can be even harder. Firstly, the estimations are based on a scope which is inherently inaccurate. Furthermore, estimating how long something will take is like looking into a crystal ball of the future. The further into the future we try to predict, the harder it is. In fact, the cone of uncertainty illustrates this. The variance on the time estimate increases exponentially. The reasons for this can be accounted for in scope variations like missed stories or new stories being added in, overly optimistic time estimations, and unknown unknowns. In Appendix E we have listed out some of the risks associated with a software project that can contribute to inaccurate time estimates.

5.6. Story Estimation

Summary

Estimating the length of a software project is notoriously difficult. As discussed in Section 5.5 on managing expectations, there are many risks involved so having a robust estimation process is essential. There are a number of different ways to go about estimations and you will find some similarities here with other methods. One key difference is that we use risk to influence our estimations. We also use some other factors that attempt to better match the estimation to reality.

Level of difficulty

Hard

Before you Start

The requirements backlog must be up to date. If you do not have a requirements backlog, follow the *Reverse-Engineering Requirements* activity described in Section 5.2. It is also recommended you have already run the *Managing Expectations* activity described in Section 5.5.

Stage

Prepare

Suggested time

1 to 4 hours depending on the size of the backlog.

Participants

Squad Lead, UX Designer, Web Engineer, Tools Engineer. The estimations must be strictly done only by the squad that will be developing the project. The Account Manager and Business Analyst are not allowed to attend this activity as they may inadvertently (or intentionally) influence the outcome.

Materials

Spreadsheet

Codebots[3]

Steps

1

Access the story estimation spreadsheet from `https://codebots.com/story-estimation`. Open it, save a copy and have a look around.

2

Fill in the user stories on the X sheet. These are the rows in the table as shown in Figure 5.7.

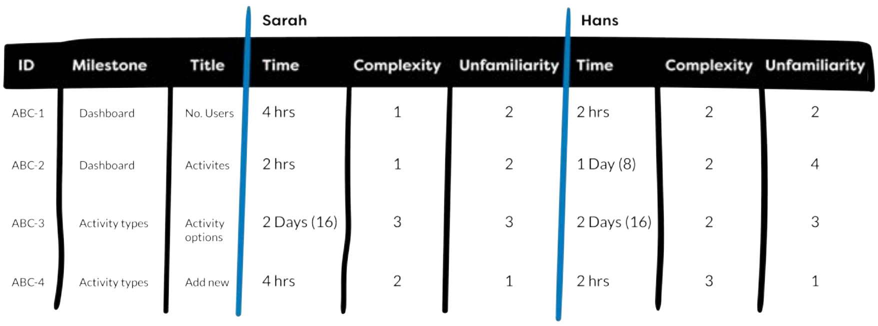
3

Put the squad members across the top as the columns in the table as shown in Figure 5.7. It is recommended a minimum of 3 team members.

4

The squad lead will choose a story to estimate. Without discussion, each squad member is to write down their estimation. The choices for the length of the story is a fibonacci-like sequence shown below:

[3]It is on the roadmap for Codebots to provide story estimations as part of the platform. Until it is available, using a spreadsheet will suffice.



			Sarah			Hans		
ID	Milestone	Title	Time	Complexity	Unfamiliarity	Time	Complexity	Unfamiliarity
ABC-1	Dashboard	No. Users	4 hrs	1	2	2 hrs	2	2
ABC-2	Dashboard	Activites	2 hrs	1	2	1 Day (8)	2	4
ABC-3	Activity types	Activity options	2 Days (16)	3	3	2 Days (16)	2	3
ABC-4	Activity types	Add new	4 hrs	2	1	2 hrs	3	1

Figure 5.7: Story estimation spreadsheet estimates

- 1 hour (0.13 days)
- 2 hours (0.26 days)
- 4 hours (0.52 days)
- 8 hours (1.05 days)
- 16 hours (2.11 days)
- 32 hours (4.21 days)
- 64 hours (8.42 days, 1.68 weeks)
- 128 hours (16.84 days, 3.37 weeks)
- 256 hours (33.68 days, 6.74 weeks)
- 512 hours (67.37 days, 13.47 weeks)

5

The next step is to assign a risk level to the story. The risk is based on unfamiliarity and complexity.

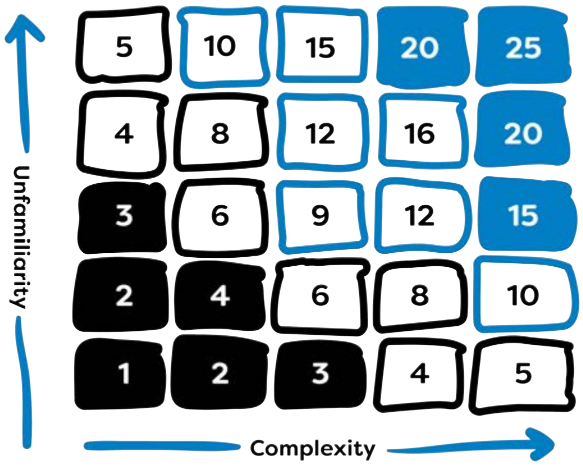


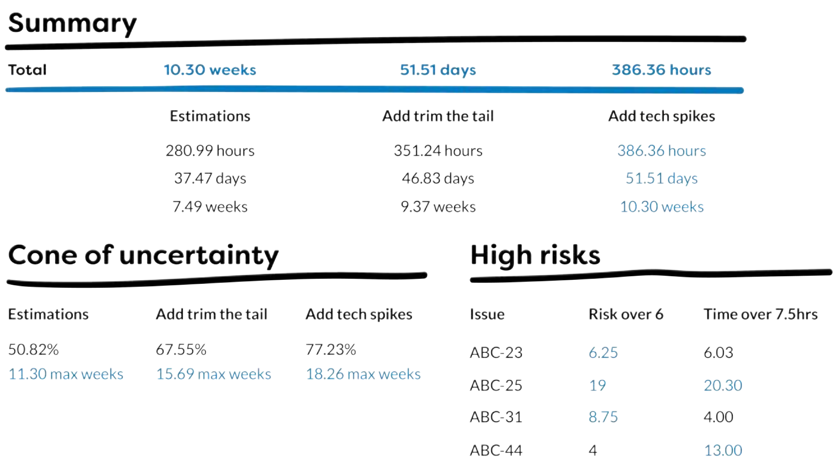
Figure 5.8: Story estimation risk matrix

6

After each squad member has their time and risk estimates, the squad lead can facilitate a discussion and the squad can listen to different justifications and change their estimations if they want. However, stick to your guns if you need to! Go back to step 4 and repeat for all the stories.

7

The last step is to analyse the summary found on the last sheet and shown in Figure 5.9.



Summary

Total	10.30 weeks	51.51 days	386.36 hours
	Estimations	Add trim the tail	Add tech spikes
	280.99 hours	351.24 hours	386.36 hours
	37.47 days	46.83 days	51.51 days
	7.49 weeks	9.37 weeks	10.30 weeks

Cone of uncertainty

Estimations	Add trim the tail	Add tech spikes
50.82%	67.55%	77.23%
11.30 max weeks	15.69 max weeks	18.26 max weeks

High risks

Issue	Risk over 6	Time over 7.5hrs
ABC-23	6.25	6.03
ABC-25	19	20.30
ABC-31	8.75	4.00
ABC-44	4	13.00

Figure 5.9: Story estimation spreadsheet summary

Justification

For any person that has dealt with computers, they know to expect the unexpected. For example, a software engineer can be doing a task and complete most of the work very quickly in a few hours, then some very strange behaviour happens and they spend the next 3 days trying to debug and figure out what the heck is going on. This is not a reflection on the person's ability; after you have spent many years in the trenches you come to the realisation that this is very complex work that carries a lot of risks. This is a major source of frustration and is one of the reasons why doing estimations can be an agonising process as it is the unknown unknowns that will inevitably spring up when the development is underway.

We use time and not a points-based mechanism to estimate an iteration. When we estimate time we do so on the assumption that the squad members responsible for completing the work have reached a standard level of proficiency. For example, a web developer is qualified on using the JavaScript MVC framework and they are able to create a custom tile, view, or component within a timeframe because that is what they are qualified to do. If a developer is faster at completing a task than what was estimated, then this time is saved on the overall project. For this reason, stories are estimated on typical qualified squad members.

5.7. Entity and Requirements Traceability Matrix

Summary

The entity and requirements traceability matrix is used to help systematically record a divide-and-conquer migration pattern (described in Section 2.4). It is possible to create a traceability matrix using a spreadsheet and use the Codebots platform to assist with the process.

Level of difficulty

Hard

Before you Start

The requirements backlog must be up to date. If you do not have a requirements backlog, follow the *Reverse-Engineering Requirements* activity described in Section 5.2. You will also need to have access to the legacy database schema(s) and get a list of all the tables.

Stage

Plan

Suggested time

Unknown, hopefully less an iteration (usually 2 weeks).

Participants

Business Analyst. Anyone else with knowledge on the legacy system.

Materials

Spreadsheet

Codebots[4]

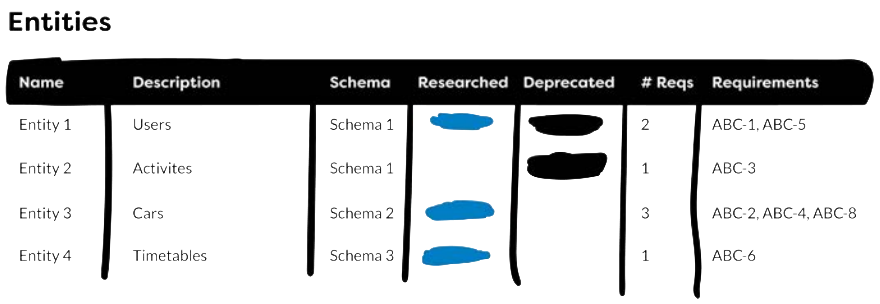
Steps

1

Access the entity and requirements traceability matrix spreadsheet from `https://codebots.com/entity-requirements`. Open it up, save a copy and have a look around.

2

Populate the entities on the *entity* sheet as shown in Figure 5.10. The entities are the tables found in the legacy database schema. It is recommended to include all of them to ensure that nothing is missed from the legacy database.



Entities

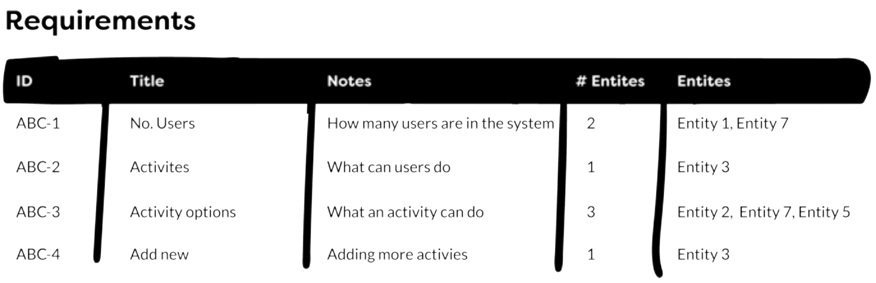
Name	Description	Schema	Researched	Deprecated	# Reqs	Requirements
Entity 1	Users	Schema 1	🟦	⬛	2	ABC-1, ABC-5
Entity 2	Activites	Schema 1		⬛	1	ABC-3
Entity 3	Cars	Schema 2	🟦		3	ABC-2, ABC-4, ABC-8
Entity 4	Timetables	Schema 3	🟦		1	ABC-6

Figure 5.10: Fill out the entities

[4]It is on the roadmap for Codebots to provide the entity and requirements traceability Matrix as part of the platform. Until it is available, using a spreadsheet will suffice.

3

Populate the stories on the *requirements* sheet as shown in Figure 5.11. The stories are the requirements and can be reverse engineered from the legacy system as described in Section 5.2.



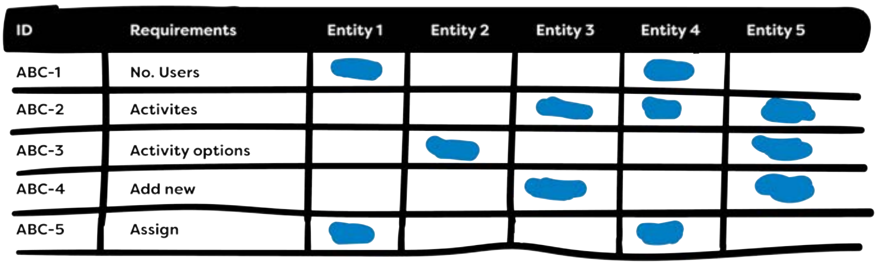
Requirements

ID	Title	Notes	# Entites	Entites
ABC-1	No. Users	How many users are in the system	2	Entity 1, Entity 7
ABC-2	Activites	What can users do	1	Entity 3
ABC-3	Activity options	What an activity can do	3	Entity 2, Entity 7, Entity 5
ABC-4	Add new	Adding more activies	1	Entity 3

Figure 5.11: Fill out the stories

4

Check the relationship between the entities and the stories on the *traceability* sheet.



ID	Requirements	Entity 1	Entity 2	Entity 3	Entity 4	Entity 5
ABC-1	No. Users	●			●	
ABC-2	Activites			●	●	●
ABC-3	Activity options		●			●
ABC-4	Add new			●		●
ABC-5	Assign	●			●	

Figure 5.12: Use the matrix to relate the entities and stories

5
Inspect the *stats* sheet to gain insight on the project.

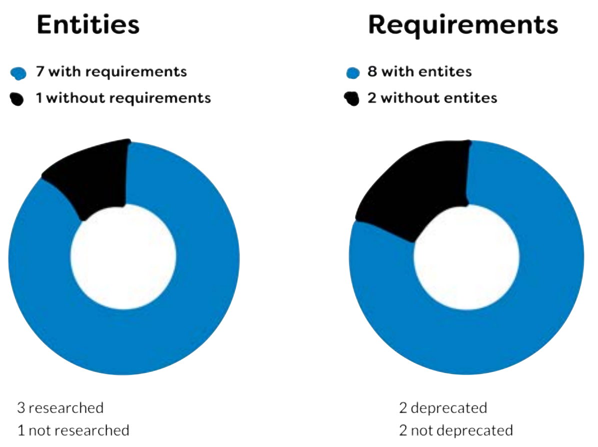


Figure 5.13: Entity and requirements traceability matrix summary

Justification

The difficulty with complex legacy migration projects is not so much the size of them, it is the systematic approach to how they are migrated over time. There will be a significant amount of effort in establishing what it is the legacy system actually does, and what parts of the organisation it is used in. The journey of continuous modernisation is to find a software/people fit so that the organisation can be agile and meet change. The traceability matrix presented in this migration kit activity goes a long way to helping the journey.

Before this activity is started, the requirements are reverse engineered from the legacy system. This activity by itself gives a target list of stories. But for large and complex systems where a big bang, or firecracker, is not feasible, the project must be broken down into a number of iterations. Our way of working is to group several iterations into a milestone and use scoping iterations as shown in Figure 5.14. Each iteration will have one or more stories based on the goals and value that will be delivered. Using the entity and requirements traceability matrix it is now possible to work out which entities must be included as part of this iteration. This shines a significant amount of light onto the project and allows a systematic migration process.

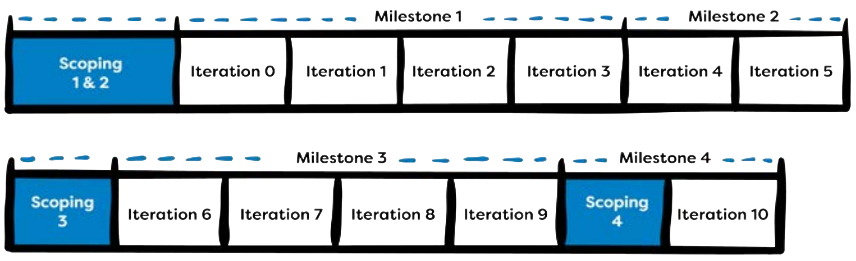


Figure 5.14: Milestones, iterations, and scoping

However, there are a few traps to look out for that can arise from this process. What happens when an entity belongs to two requirements that are in different iterations? There are a few ways to work around this. First, does it make sense for the requirements to be shifted into the same iteration? If it does, then shift the requirements so they are in the same iteration. If it does not, you can leave the requirements in seperate iterations but the isolation checker will fail in the divide-and-conquer migration pattern. Recall that the isolation checker is a safe guard that ensures the data integrity of the new application (refer back to Figure 2.4). To make sure the isolation checker is going to function as expected, any releases that include an entity must include all of the requirements that relate to that entity. A nice way to ensure this is to use milestones to group iterations that cover all the requirements for the entities being migrated to the new application.

Furthermore, the entity and requirements traceability matrix can be used to help decide on the boundaries for the new microservice applications. We have included another approach called the bubble context and anti-corruption layer found in Section 5.11. A different approach is to base the new application suite on the natural boundaries that arise from the matrix. For some legacy systems, the matrix can help reveal a new set of microservices relatively quickly, but for some more complex legacy systems the new set of microservices is not as obvious. It is our intention to present some techniques in a future version of this book.

5.8. Database Migration

Summary

A database migration is a typical starting point for a new application on the Codebots platform. The steps described next will migrate your legacy database to a new application architected using microservices.

Level of difficulty

Moderate

Before you Start

You will need a copy of the database. If the database type you are migrating from is not yet supported, try exporting your database to a database schema in standard SQL.

Stage

Migrate

Suggested time

60mins depending on how complex security and administration you will be implementing.

Participants

Product Owner, Business Analyst. Possibly a technical team for database types not yet supported

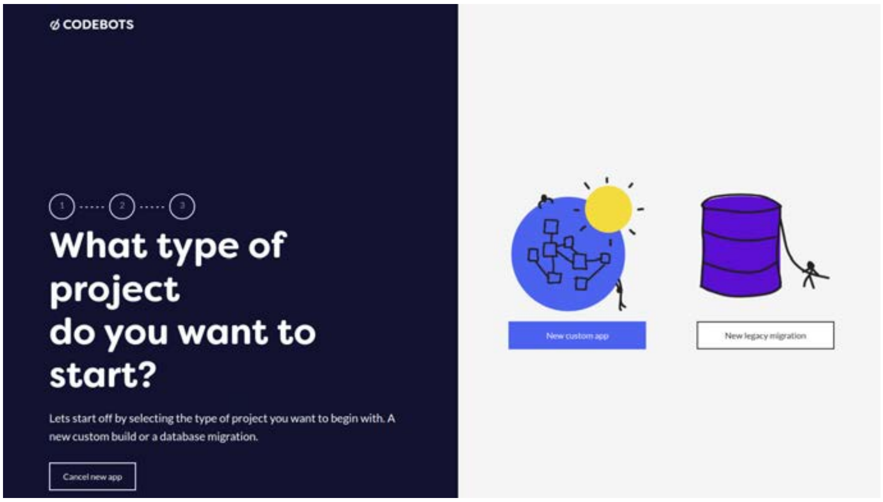
Materials

Codebots

Steps

1

Create a new App on the platform and choose new legacy migration.



CODEBOTS

1 2 3

What type of project do you want to start?

Lets start off by selecting the type of project you want to begin with. A new custom build or a database migration.

Figure 5.15: Choose new legacy migration

2

Inspect the entity diagram as described in Section 4.2 and repair the database if needed.

3

Setup the security groups using the security diagram described in Section 4.4 and lock down the Application Programming Interface (API).

4
Setup the CRUD administration pages in the application using the UI diagram as described in Section 4.3.
5
The last step is to migrate any extra *business logic* and the process to follow is described in the justification. The business logic of the legacy database can usually be found in multiple locations including:

- Forms (such as Microsoft Access forms).
- Stored Procedures.
- Views.
- Custom Code.

Justification
There are some code analysis tools that could prove helpful to find the important sections of the code, but in our experience using automated tools to migrate between languages is not efficient due to the *garbage in, garbage out* inescapable truth. So, roll up your sleeves and try to understand the code. Follow the process of reverse engineering the requirements as describe in Section 5.2 and look at using behaviours to cover the business logic (see Appendix C for a list of behaviours). If the business logic is not covered by a behaviour, then you will need to create custom code as shown in Figure 4.2.

5.9. Spreadsheet Migration

Summary

Spreadsheet migration is a common type of migration as there are many limitations that prevent spreadsheet from being scalable. In most cases spreadsheet consists of a way for adding, modifying or removing of data with certain fields being dynamic and calculated through the use of formulas.

Level of difficulty

Hard

Before you Start

You will need to be confident on how to design a database schema. You will be required to identify the entities, attributes, and relationships of the data found in the spreadsheet and create an entity diagram to match.

Stage

Migrate

Suggested time

Around 60 mins for basic spreadsheet and two days for complex spreadsheets. It could be longer depending on how messy the spreadsheet is.

Participants

Business Analyst, Product Owner. Possibly a technical team if the spreadsheet contains formulas.

Materials

Codebots

Steps

1

Start a new empty project on the platform.

2

Open the entity diagram as described in Section 4.2. It will be blank.

3

Identify the first entity for your spreadsheet. On one of the sheets, you will have some tabulated data where the rows of the data and the *things* on that sheet. Whatever that thing is, is your entity. Name it.

4

For each column within your spreadsheet we will need to create a new attribute for the entity. Be mindful that you should try to retain the cell formatting by matching with a corresponding attribute types or validation pattern.

5

Repeat 3 and 4 for as many entities you can identify in your spreadsheet. Then add the relationships between the entities. This is a normal part of database schema design. Your options are one-to-one, one-to-many, and many-to-many.

6

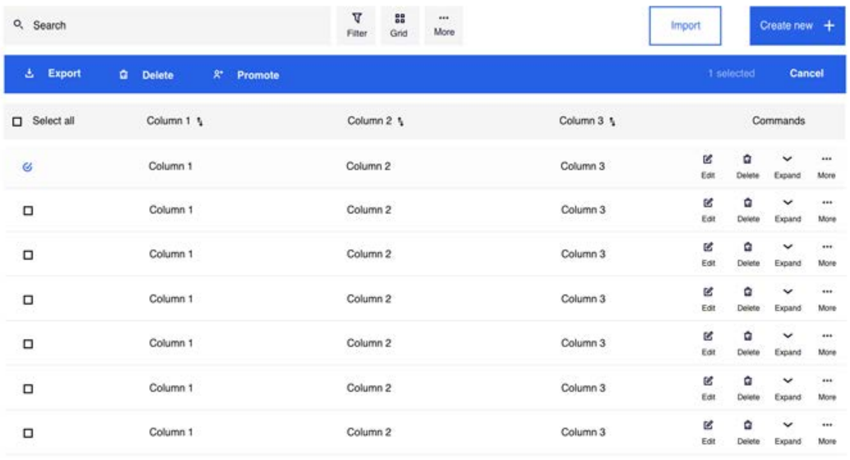
Once you have the entity diagram completed, setup the CRUD administration pages in the application using the UI diagram as described in Section 4.3.

7

Command your codebot to write the code and deploy the application. You should now be able to access the application in a beta environment.

8

On the CRUD page, you will be able to import the data. Click the import button as seen in Figure 5.16. On the import screen you will be given an option to download a blank spreadsheet that have the columns and some explanations inside it. You will need to manually copy and paste from the original spreadsheet into this spreadsheet to upload the data.



☐ Select all	Column 1	Column 2	Column 3	Commands
✓	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More
☐	Column 1	Column 2	Column 3	Edit Delete Expand More

Figure 5.16: Import and Export UI

9
For the complex formulas and functions found in the spreadsheet, we recommend following the Reverse Engineering Requirements migration kit activity found in Section 5.2. Once you have worked out what the true intention of the complex behaviour is, then you can move forward and create a great UX.

Justification

Spreadsheets are a common tool that many organisations use for all sorts of various processes. They are great because you can get a quick win and solve relatively complex problems without needing knowledge of how to program. To get more complex spreadsheets however, you will need to be able to use formulas and functions, which is a type of programming. There are a number of problems that organisations run into when you spreadsheets have out lived their usefulness.

One of the weaknesses of spreadsheets was getting concurrent users working on them. In the old days, the spreadsheets would be emailed around and it would be easy to lose track of which was the latest or merge in changes across different users. More recently, there are now online spreadsheets such as Google Sheets (`https://www.google.com/sheets/about/`) that help with this and provide multi-user access. Still the biggest problem that remains is the UI. Asking the people of your organisation to log in and update a spreadsheet for a particular business function end up being a poor UX. Furthermore, spreadsheets become unmaintainable quickly once formulas and functions are added in due to the openness and changes that can be made. It is pretty much a free-for-all. There are some more advanced online tools that allow you to build a UI on top of a spreadsheet like AppSheet (`https://www.appsheet.com/`). Make sure you look at the total cost of ownership for no-code platforms like this and carefully consider who owns the source code of your application.

5.10. PDF Migration

Summary

Migrating a PDF form to a web-based form is a common task. Most of the configuration of the new form is done in the application and not in the model. The reason for this is to ensure that the administrator of the application has the control to make changes to the forms without the need for further deployments. So, the first few steps of the migration are enabling the forms in the diagram editor and deploying the application. Once this is completed, the majority of the work is then done in the application except when custom tiles are required. See Figure 5.17 for the PDF migration workflow to help visual the process.

In the justification section, there are more details on how to use the functionality of the forms such as slides, skip logic, show logic, and what tiles are available. There is however an important concept that should be understood. When you are migrating a PDF form and you come across a section where none of the standard tiles match your desired functionality, you will need a custom tile.

Level of difficulty

Moderate

Before you Start

All forms in a business follow a different process to make sure that they are handled correctly. In order to make sure that the migration of these forms is successful, it is important to understand every step

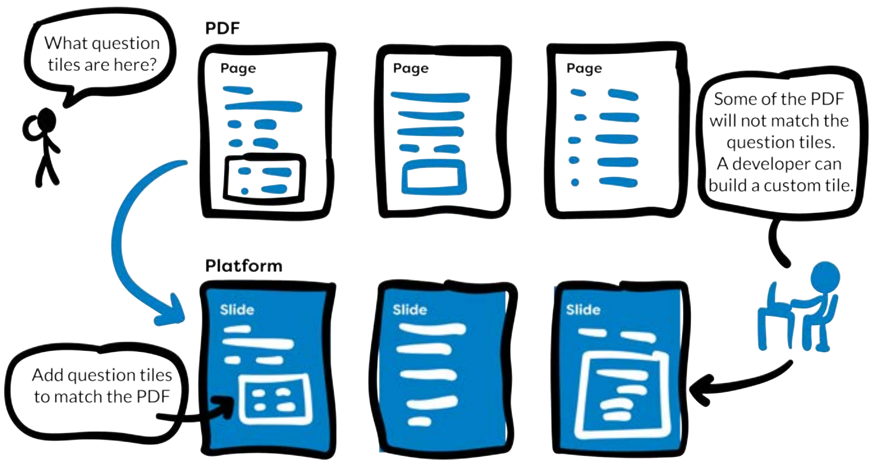

What question tiles are here?
PDF
Page
Page
Page
Some of the PDF will not match the question tiles. A developer can build a custom tile.
Platform
Slide
Slide
Slide
Add question tiles to match the PDF

Figure 5.17: PDF Migration Workflow

a form goes through, from how it is processed, to who is involved, to the different resources that a user may require access to in order to fill it out. If you don't know all of the details yourself, make sure you have access to someone you can consult with for each form.

Suggested time

60mins per PDF form (depending on the size and complexity of the PDF).

Stage

Migrate

Participants

Business Analyst, Product Owner. Possibly a technical team for custom tiles.

Materials

Whiteboard, or potentially printing and post-it notes with lots of wall space.

Steps

Planning your Form

There are a number of ways in which you can plan your form. If you want to keep it exactly the same, you could print out the form (or use a PDF editor) and draw squares around every ”input” you want to be included. This is essentially any place where you need the user to write in some details or select an option. Later, these will be recreated using question tiles.

If you want to reorganise or reconfigure the form, there are a large variety of options available to you; you could use post-it notes, a whiteboard, or even cut out the questions in the form and use blu-tack. It is important that you work your way through the form, making note of the key sections in the form and write them down with the questions inside. At this point, you can rearrange the sections and questions into whatever order you desire. The sections allow you to keep the related questions all together so the form follows a logical order.

Even if you aren’t reordering your questions or altering the form, it is important that you familiarise yourself with the form itself so you can be confident that what you are building is correct.

1

Enable forms in your entity diagram, and include form display tiles in the interface builder anywhere you want one of these forms to go (the interface step can be done later, though it will require a re-deploy).

2

Deploy your application and open up the backend. If you added forms successfully, it should now be available as an option in the menu. Navigate to it and create a new form.

3

Enter the details of the form, and configure the display settings in a way that suits your application and the requirements of your form. Everything will autosave as you go. In this page, you can also select the location this form is displayed in, according to the locations you specified in step one (if you did use the UI builder).

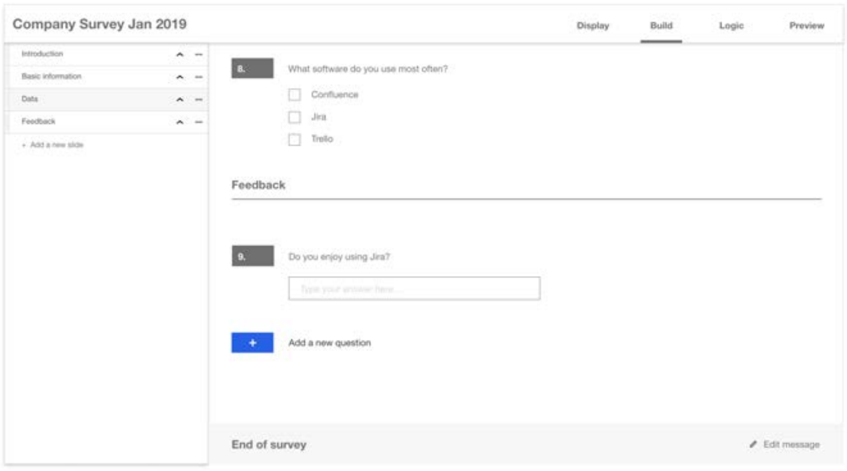
4

Go to the build tab of the form builder. In this screen you can add and edit question tiles, control their options, reorder them, and manage your slides (see more information on slides in the justification section). Following your plan, start adding in questions (and configuring their validation if necessary).

5

Once complete, you can move to the logic tab if necessary, or you can preview the form you have built.

6
If you are happy with everything, you can publish your form and it will become available on the frontend of your application (provided you completed the interface part of step one).
7
Repeat those steps as necessary until you have completed all of your forms.



8. What software do you use most often?
- ☐ Confluence
- ☐ Jira
- ☐ Trello

9. Do you enjoy using Jira?

Figure 5.18: Backend Form Builder

Justification
PDF forms can be used in any number of ways within a business. From being used internally to capture reports, on the company's website for people to fill out and email back, or printed for customers to fill out and post back to you as snail mail, they have

become deeply embedded in the processes of a lot of companies. No matter how they are included in a business process, the use of them requires a whole workflow to fill them out, submit and then process them in the correct way. In a world where customers are used to being able to fill out forms online and have them submit instantly, the process of filling out PDFs becomes an inconvenience they want to avoid. For this reason, it is important that we ease this process as much as possible. By migrating the workflow over to be a web-based system, the users are able to complete the process with more efficiency and the risk of invalid values becomes significantly smaller.

Submissions

Forms can be filled out multiple times by multiple users. Each time a user submits their form, a new submission is saved in the database. All submissions received for a form can be viewed in your application's backend.

Versions

Versions are in place to allow you to edit and create forms, without users seeing it mid-build. It allows you to control when your latest changes are made available to the public so you can edit a form, save and walk away, without the partially edited version being shown to users until you are ready. This is quite handy if you regularly need to update a form with new questions or standards, or if you want to reconstruct an existing form. Versions are automatically made when you edit a form which has been published, though you control when your new version is published. You can see previous versions of a form from the backend of your application.

Slides

Slides are a way of introducing logical grouping to your questions, and are used to give you control over how your questions are displayed. They can be thought about in a similar way to Powerpoint Slides, in that you use them to display a few pieces of information before moving onto the next one. Using the display options, you can also configure whether they display as seperate ”pages”, or if they all show on the same page but with a distinct break between the sections. They can also be helpful when building the logic behind a form as they allow you to skip large sections at once.

Tiles

Tiles are how you add questions into the form. There are a large variety of types available, some of which allow you to switch between the types (i.e. dropdown, checkboxes and radio buttons). You can add, remove and duplicate tiles, in addition to being able reorder them.

Logic

There are two types of logic which are available inside a form: skip or show. The logic is based around how a user answers a given question, and runs the logic when they answer. *Skip logic* allows you to show a question normally, but hide it occasionally. *Show logic* means that a question is normally hidden, and is shown only if a question is answered a certain way. You can choose whether the logic is applied to an individual tile or a whole slide.

Due to the nature of both kinds of logic, they can be used to recreate the other type (though it can make things complex). It is up to you around how you choose to approach this, though we do recommend thinking about this first before you start making the logic. This should be considered while you are still in the planning stage so you can draw arrows between things and play out some scenarios to make sure you understand everything.

Once your logic is built, you can test it using the preview section of the form builder. The more complicated your logic is, the more important it is that you test it thoroughly. It is possible to make it impossible to finish a form if the logic is incorrect. For that reason, the preview tab does have a reset button in case you get stuck, which takes you back to the start of the form.

5.11. Bubble Context and Anticorruption Layer

Summary

DDD [8, 9] divides a complex domain up into multiple bounded contexts and maps out the relationships between them. From these these bounded contexts, a single one is selected to become the bubble and its relationships between the other bounded contexts are analysed in detail. These relationships become the basis for the anticorruption layer where any conceptual objects from the legacy model are translated before being utilised by the selected bounded context.

Level of difficulty

Moderate

Before you Start

An application domain contains many subdomains that represent its inner workings. This is especially true for many legacy systems where the software has simply grown over time to address changing requirements. Sometimes a legacy system may be made up of a series of smaller applications with potentially complex relationships between them. Each of these applications could be considered subdomains of the whole and, in their own right, domains with their own subdomains.

The existence of these domains and subdomains allows for a measured approach to migration that addresses each component or subdomain in turn without corrupting the new system with existing concepts and ideas. Before starting this activity, a good understanding of the existing systems domain is important.

Suggested time

60 minutes per bubble context (assuming the complexity of its iterations to the system as a whole does not go beyond three distinct connections).

Participants

Business Analyst and a technical team.

Materials

Whiteboard or butchers paper.

Steps

1

Break your complex domain down into a series of bounded contexts with the relationships between them mapped out. For this we make use of something called *concept mapping* [5].

2

From theses select your context in which you will create your bubble from. It is ideal that this is not too complex but provides enough intricacy that there is value in migrating it.

3

Now that we have our selected bounded context, we can begin to create our new model. Ensure that this model is built based on the requirements of the system and not on the legacy system. This is

[5]For more information see Eric Evans keynote at Explore DDD 2017 `https://www.youtube.com/watch?v=kIKwPNKXaLU`

your opportunity to improve and remove redundancy.

4

Have a look at the relationships between your selected bounded context and the others in your domain. List these in detail taking special note of the conceptual objects involved (i.e entities, services, data structures, protocols etc).

5

Given this list, explore how they would be represented within the bubble context and update your model accordingly. Write this down.

6

With with these details we can now start building out what is required by our anticorruption layer. This will include any services, translators and adapters. Ask the questions,

1. What data do we need?
2. How does this data differ between the two contexts.

Once we have built out our anticorruption layer and our bubble context we are ready to put this into action.

Justification

A highly important part of migrating a legacy application is ensuring that the new application does not end up inheriting the technical debt apparent within the existing legacy model. It is almost equally important for a successfully migration that any risks associated with the project are minimised and the overall exposure is reduced to maximise success. To achieve this, we use two domain driven design concepts. The bubble context and the anticorruption layer.

To have a bubble context is to have established a small bounded context through the use of a an anticorruption layer. This small bounded context reduces the risk and complexity when migrating large legacy systems through the breaking down the problem into manageable pieces. The anticorruption layer protects this smaller piece of the larger system from corruption from the old legacy system and allows it to be developed independently.

Through independence, many of the old concepts or models that may exist in the legacy system can be adopted or removed completely as required for the new system, without any existing dependency being broken.

While being developed independently and in isolation, the new migrated system that exists within our bubble can still interact with the legacy system and as such can still be integrated as a useful part of the whole while still maintaining its own model.

This is achieved through the use of the anticorruption layer. The anticorruption layer provides a series of services, facades, adapters and translators to transform the data from the legacy context into the new migrated context contained within our bubble and often vice versa maintaining the connection and the integrity of both systems (see Figure 5.19). This separation of the two contexts prevents either context from morphing to mirror the other and thus allowing the new model to remain strong.

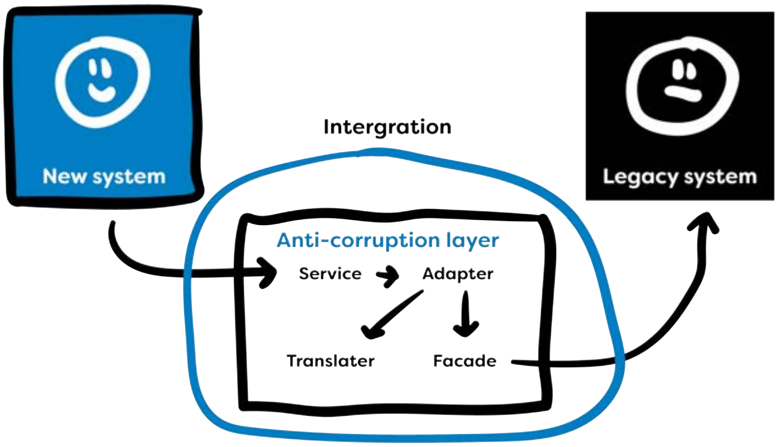


Figure 5.19: Anticorruption layer

An anticorruption layer can be utilised to bridge multiple bounded contexts with the same benefits found when using the single context contained within our bubble [6]. This assists in the migration and integration of mutiple parts of the system.

While the purpose of the bubble context is to maintain separation from the legacy system, bubbles can pop. When this occurs the context maintained within the bubble can be integrated back into the existing system seamlessly through the use of our anticorruption layer.

[6]For more detail on the how the bubble context and anticorruption can be useful tools in legacy migration watch Eric Evans talk on Domain Driven Design Strategies for Dealing With Legacy Systems `https://www.youtube.com/watch?v=OTF2Y6TLTGO`

Once an application has started to be broken down into it’s component contexts, the concept of microservices start to make more sense as the reduction in risk can be maintained through the usage of a microservice architecture (Section 3.3).

As each bounded context of the system is transformed into a bubble context, they can be integrated back into the system as a microservice.

6. Conclusion

“True knowledge exists in knowing that you know nothing.” Socrates

Codebots is both a methodology and a set of technologies. It is a methodology because it lays out an approach to how a codebot evolves over time. As long as time moves forward and the team follows the methodology, a codebot will improve. It is a set of technologies because people can interact with a codebot to write a full-stack software applications. A codebot is designed to be complimentary to other software frameworks, methodologies and applications, not a replacement. Similarly, a codebot is not a replacement for software engineers. A codebot does the heavy lifting of the project and can solve common business patterns. It is up to the humans to use their creativity to solve the truly complex problems that a machine cannot. A codebot helps people build software faster, of better quality, and with greater reuse.

The hypothesis presented in this book is that business agility can be positively impacted through a process of continuous modernisation. Continuous modernisation is a strategy for legacy systems to ensure the *software* fits the *people* using it. The problems faced include technical challenges around the *software* and cultural challenges around the *people*. When a software/people fit is found, momentum is gained by the organisation and business agility is increased.

All software systems are legacy systems the moment they are deployed and people start using them. As time progresses, the legacy system suffers from software entropy as technical debt, knowledge loss, and changing markets all contribute to the negative impact on the organisation. Since time cannot be stopped, legacy systems are an inevitable consequence that organisations must address.

Agile software teams are acutely aware of technical debt and dedicate time to cleaning up and staying on top of the problem. However, the older the software, the more problems it suffers from. Some really old systems no longer have any relevant documentation, the original developers have left, and the market has changed so significantly that the software is no longer fit for purpose. Furthermore, the software can entangle itself with other software systems and the people can become accustomed to its use. So, the updating or removal of the legacy system becomes a significant problem, both technically and culturally.

This is a complex problem with many contributing factors. There is no magic wand to make it disappear. However, it is possible to put strategies and tactics into play that help minimise the risks. A strategy is the overarching plan. Tactics are specific actions undertaken as part of the strategy.

The strategy for continuous modernisation is a three-phase approach. Phase 1 is designed to enable organisations to start on their journey of continuous modernisation. During phase 1 the first legacy system will be migrated. Typically, this will be a smaller system with small impact. Start with the low-hanging fruit. Phase 2 is to connect this newly migrated legacy system with other systems, such as downstream reporting tools (this will really unlock the data). It can also include developing the new system further to find a better software/people fit. Phase 3 is to continue onto other legacy systems within the ecosystem. During the earlier migrations, you should concentrate on the satellite systems to isolate the larger monoliths before taking them on.

While executing on this strategy we recommend five tactics, a migration kit, and hopefully by now you can see the benefits of using a codebot. The migration kit is a set of activities that can be used during a legacy migration project. They are the arrows in your quiver and each activity is self-contained and may or may not be used on a project. The five tactics should be used on every project. These five tactics are aimed at enabling the organisation to gain momentum and increase business agility. The five tactics are:

1. Use science to drive innovation at a pace in step with the organisational rhythm (Section 3.1).

2. Enable the community around your organisation. A strong community builds a defendable moat around your organisation (Section 3.2).

3. Build small loosely coupled microservice applications emphasising a separation of concerns (Section 3.3).

4. Increase the visibility and control of the entire software ecosystem by lifting the fog of war (Section 3.4).

5. Use models to capture knowledge and enable control through the use of codebots to do the heavy lifting on software projects (Section 3.5).

The overall result of these tactics is to create a software ecosystem where the legacy systems are smaller, visible, and loosely coupled, so that they are more able to be modernised in the future. There is a perfect opportunity to set a mode of continuous modernisation rolling forward alongside the agile business. As organisations embrace the mantras that underpin agile and work out their own hybrid approach that works for them, it is possible to embrace continuous modernisation. Even organisations that are not agile yet, or starting the journey, this can be a good starting point. By making time your ally, you are able to set forward a process of continuous modernisation that enables the people to carry forward with empowered teams. This becomes the role of the servant leader and can ultimately increase your organisation's health and give you a competitive advantage.

Knowledge is power and a main contributor to legacy systems is the lack of understanding about what business agilitythe system is. It is possible to create a model of the legacy system and have a codebot use the model to write significant amounts of a new target application. The model is part of the knowledge representation of the application. But there is a bigger benefit that comes the next time it's time to modernise that legacy system, you have already captured the knowledge from the last migration effort so you are in a much better position. Another way to think about it is starting at the end point and working backwards. What would be the best position to be in at the start of a legacy migration project? The answer is a clear understanding about the legacy system, why it was built, what other systems it integrates with, etc. So, to make sure we have the knowledge, let's capture it in a model that is as important as the source-code itself. This knowledge is power because you are already on the front foot to modernise the system without complex and difficult work around reverse engineering the legacy system. To get ourselves into this ideal position of knowledge, we must capture that information in today's legacy migrations projects. This is the best chance we have at breaking the cycle of legacy and our insanity.

There is cautious optimism around using a microservices architecture. The results so far have been positive. However, there will be unintended consequences and only allowing more time to play out will reveal them to us. We can however look at some other areas of software that have embraced similar principles as microservices, such as the principle of separating concerns through the use of API. For example, in object-oriented programming there is the concept of inheritance. A sub class can inherit from a base class and this can happen many times creating long dependency chains. This has resulted in a phenomenon called the *fragile base class problem*. Basically, the long inheritance chains make changes to the base class difficult because of the dependencies. It is possible to work around it with backward compatibility and well-defined interfaces, but is there a *fragile microservice problem* on the horizon we do not know about yet?

The next series of experiments that we are conducting and will publish in the next version of this book will shed light onto a few areas. The first is around what business agility actually mean? Most people acknowledge, and agree, that being ready to meet changes in market conditions is a good idea. But how do we measure the agility of a business? There is some good research that has been done in manufacturing around this and we will be gathering up more evidence and integrating it with our hypothesis. This leads into another experiment we are conducing around the fog of war. Again, we all know it exists, but how do we effectively manage it? As all good scientists do, we can admit when we don't have a theory that matches observation, however we are going to investigate this phenomenon and further our evidence to support the hypothesis.

In this book, we are not claiming to provide all the answers. Like all good scientists do, we acknowledge the things we do not know and formulate hypothesises and experiments to add to the body of knowledge on the subject. It is our intention to update this book frequently and send you a new copy each time we do. The migration kit will continue to expand and please feel free to send us any learnings that you find on your path of continuous modernisation.

Glossary

access database is another name for Microsoft Access (or MS Access), an information management tool that enables managing related information efficiently and analysis on large amounts of information. 38

actuator is a component of a machine that is responsible for moving and controlling a mechanism or system. 85, 86, 89, 113

Agile sprint is one time-boxed iteration of a development cycle. 99

attributes belong to an entity and describe characteristics such as name, height, DOB for an employee entity for example. 93, 153

backward compatibility refers to a hardware or software system that can successfully use interfaces and data from earlier versions of the system or with other systems. 178

black-box is a device, system or object which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings. 73, 74, 86, 87, 97

blockchain is a growing list of records, called blocks, that are linked using cryptography. 64

bounded context [8] is the a description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable. 166, 167, 168, 169, 170, 171

brownfield project is used in many industries, including software development, to mean to start a project based on prior work or to rebuild (engineer) a product from an existing one. 42

burn-down charts is a graphical representation of work left to do versus time. 124

business agility allows organisations to respond rapidly to internal and external changes. 18, 19, 20, 21, 22, 23, 25, 28, 33, 58, 61, 64, 72, 82, 174, 176, 177, 179

client-side is a term referring to the part of a web application running on the user's computer (also known as front-end). 70

cone of uncertainty a concept which shows how uncertainty about a thing increases over time exponentially from the point of last evaluation. 132, 133, 134, 135, 137

context is the setting in which something appears that determines its meaning. A model can only be understood in a context. 21, 22, 24, 27, 70, 122, 130, 149, 167, 168, 169, 170, 171

development target is the the source code in the target application that will be deployed to the end users. 97, 101, 102, 103, 105, 106, 108, 110, 112

domain is a sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.. 12, 13, 77, 78, 79, 91, 117, 118, 121, 166, 167, 168

entities are one thing that is modelled from the real world and corresponds to a table in the database. 26, 93, 94, 96, 145, 146, 148, 149, 153, 154, 168

entity diagram the diagram which is used to model entities. 91, 93, 151, 153, 154, 161

epics is a high-level theme or feature that is used to categorise related stories. 118, 120

fail-over a pre-configured backup which can be switched to, typically automatically, in the event of a failure. 70

field trials are an experiment to demonstrate just how much value Codebots adds to software projects. 90, 112

full-stack applications these are applications which utilise or are comprised of a client-side and a server-side. 100

hybrid mobile-apps mobile applications which are built on a framework which enables them to be compiled and deployed to the most common platforms. 70

inheritance a term in software which implies that one thing inherits the properties and behaviours of another thing. 178

iteration a set period of time during which the project team aims to deliver value to the customer. 23, 59, 62, 63, 65, 99, 101, 103, 104, 119, 120, 143, 144, 148, 149, 167

Johari Window a technique developed by psychologists Joseph Luft and Harrington Ingham that helps someone better understand their relationship with themselves and others. 133

lean enterprise is a practice focused on value creation for the end customer with minimal waste and processes. 21, 22

lean enterprise is a scientific approach to creating and managing startups and get a desired product to customers' hands faster. 21, 22, 23

legacy system any software system that is being used in an organisation. 6, 9, 12, 18, 19, 20, 23, 24, 25, 32, 36, 38, 39, 40, 41, 42, 43, 46, 47, 49, 50, 51, 56, 61, 67, 70, 71, 72, 73, 74, 78, 81, 90, 118, 119, 122, 144, 146, 147, 148, 149, 166, 167, 169, 170, 174, 175, 176, 177, 178

meta-model a model which is used to define another model. 76, 77, 78, 79

microservices a software development technique: a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. 32, 68, 69, 70, 72, 94, 98, 149, 150, 171, 178

model an abstract representation of something. 39, 47, 58, 64, 73, 74, 75, 76, 77, 78, 79, 80, 82, 89, 91, 92, 99, 109, 110, 122, 137, 158, 166, 167, 168, 169, 176, 177, 178

modeller a user who uses a diagram to model their application. 93, 95, 96

monolith is a large scale application that tends to have tightly coupled code and is affected by an inability to change quickly. 37, 69, 70, 81, 176

no-code platform provides it's user with a means to build applications with no coding experience. 157

offline capability a capability of the application which enables it to function (to varying degrees) without an internet connection. 70

OpenAPI a specification (originally known as the Swagger), for machine-readable files for describing, producing, consuming, and visualising RESTful web services. 32

pattern a general, reusable solution to a commonly occurring problem within a given context in software design. 1, 31, 32, 43, 45, 46, 47, 48, 49, 51, 70, 78, 81, 97, 98, 102, 105, 112, 113, 122, 144, 149, 154, 174

platform is found at `http://codebots.com` and where the users create and manage their projects. 30, 76, 80, 82, 86, 91, 92, 118, 119, 124, 139, 144, 145, 150, 154

protected region is a designated area within any Codebots applications that is protected from being overwritten by the bots. 87, 101

relationships are an association between two entities. The standard relationship types include one-to-one, one-to-many, and many-to-many. 93, 153, 154, 166, 167, 168

relationships is a type of software testing that aims to confirm that a recent changes to the application haven't had unforeseen affects on existing functionality. 108

requirements define functions to be performed by the system, performance measures of the system and its functions, and constraints that are imposed on the system. Requirements are defined for all fundamental inputs to an application as necessary. 14, 16, 28, 46, 49, 74, 78, 80, 86, 98, 100, 101, 106, 107, 108, 110, 115, 116, 118, 119, 121, 122, 138, 144, 145, 146, 147, 148, 149, 161, 166, 167

satellite systems are smaller application systems that surround a larger applications. 176

schema in the context of a database, it is the organisation of data as a blueprint of how the database is constructed. 43, 46, 74, 92, 93, 144, 145, 150, 153, 154

scrum master facilitates the agile process, ensure the team lives up to the values and be a servant leader to it's project team. 41, 117

security diagram is the diagram which is used to model security. 91, 96, 151

server-side is a term referring to the part of a web application running on the server or not on the user's computer (also known as back-end). 30, 70

shadow IT also known as Stealth IT or Client IT, are information technology systems built and used within organisations without explicit organisational approval. 71

show logic is added to a slide to determine if the slide should be shown or not. 158

skip logic is added to a slide and if evaluated to true, will skip to the configured slide. 158

slides are the sequence of pages found in a form. Slides can be grouped into phases and a slide will have one or more tiles to display information to the user. A slide can have skip logic and show logic to allow configuration on the order the slides could be presented. 158, 161

software ecosystem refers to the entirety of an organisations software systems and how they interact with each other. 56, 58, 71, 72, 82, 176, 177

Spring in the context of software: an application framework and inversion of control container for the Java platform. 30, 70, 98, 113

stored procedures is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs. 70

technical debt is the resulting work required throughout the development process, which can come in the form of bug fixes, code refactor or writing documentation. 71, 105, 168, 174, 175

test execution tool is used to record a test procedure, then running that procedure and capturing the results. 110

test coverage is defined as a metric in software testing that measures the amount of testing performed by a set of test. 30, 31

testing target is the the source code used to test the target application. 30, 106, 108, 110, 112

tests scripts are used for capturing the steps to be run when performing a test. 110

tiles are the basic building blocks of the development target and are a fundamental way to deliver functionality using microservices. 94, 95, 158, 159, 160, 161, 164

traceability matrix is table used to trace the state of the relationship between two isolated collections of things. 74, 106, 107, 108, 144, 145, 147, 148, 149

two-factor is a method where by a second layer of authentication, on top of a password, is used to authenticate a user. 104

user story is a fine-grained requirement, written from a user‚Äôs perspective. 99, 101, 120

user groups are a method of grouping system users for the purpose of application access or security control. 96, 117

validators are placed on attributes to ensure that the data entered conforms to a validation rule. 93

valley of death refers to the difficulty of dealing with the various complexities of growing a business. Can occur at different stages of a business. 20

views in the context of client-side development: an isolated piece of logic which contains the layout, logic and interaction events, often comprised of other smaller views. 91, 94

waterfall is a development process which emphasises linear sequential flows of development, where one piece of functionality can not be started until the previous has been completed. 30, 63, 132, 135

white-box when a machine, system or application whose internal structure or processing is known in addition to the knowledge about its inputs, outputs, and the relationship between them. 74, 87, 88

Acronyms

AAA	Authentication, Authorisation, and Auditing
AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
CI	Continuous Integration
CRM	Customer Relationship Management
CRUD	Create/Read/Update/Delete
DDD	Domain-Driven Design
DIKW	Data, Information, Knowledge, and Wisdom
DSL	Domain-Specific Language
KM	Knowledge Management
M2M	Model-To-Model
M2T	Model-To-Text
MBT	Model-Based Testing
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development

ML	Machine Learning
MVC	Model View Controller
MVP	Minimum Viable Product
NLP	Natural Language Processing
PMF	Product/Market Fit
SPL	Software Product Lines
SQL	Structured Query Language
SUT	System Under Test
UI	User Interface
UAT	User Acceptance Testing
UX	User Experience
VR	Virtual Reality

Bibliography

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” in *5th International Conference on Learning Representations*, 2017.

[2] K. Beck, *Implementation patterns.* Pearson Education, 2007.

[3] C. P. Bonini, *Simulation of Information and Decision System in the Firm.* Prentice-Hall, 1963.

[4] P. Clements and L. Northrop, *Software product lines.* Addison-Wesley, 2002.

[5] E. Escott, P. Strooper, P. King, and I. J. Hayes, “Model-Driven Web Form Validation with UML and OCL,” in *Current Trends in Web Engineering*, ser. Lecture Notes in Computer Science, A. Harth and N. Koch, Eds. Springer Berlin Heidelberg, 2012, vol. 7059, pp. 223–235.

[6] E. Escott, P. Strooper, J. Steel, and P. King, “Integrating Model-Based Testing in Model-Driven Web Engineering,” in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, 2011, pp. 187 –194.

[7] E. Escott, P. Strooper, J. G. Süß, and P. King, “Architecture-Centric Model-Driven Web Engineering,” in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, 2011, pp. 106 –113.

[8] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[9] ——. (2013) Getting started with ddd when surrounded by legacy systems. [Online]. Available: http://domainlanguage.com/wp-content/uploads/2016/04/GettingStartedWithDDDWhenSurroundedByLegacySystemsV1.pdf

[10] T. Falls. (2016) Keen io's community commitment curve and the keen onion. Keen,io. [Online]. Available: https://speakerdeck.com/timfalls/keen-ios-community-commitment-curve-and-the-keen\-onion

[11] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] J. A. Goguen and C. Linde, "Techniques for requirements elicitation," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*. IEEE, 1993, pp. 152–164.

[14] Y. N. Harari, *Sapiens: A brief history of humankind*. Random House, 2014.

[15] V. Harnish, *Scaling Up: How a Few Companies Make It... and Why the Rest Don't*. Gazelles Incorporated, 2014.

[16] J. Humble, J. Molesky, and B. O'Reilly, *Lean Enterprise: How High Performance Organizations Innovate at Scale*. O'Reilly, 2015.

[17] M. T. Jones, *Artificial intelligence: a systems approach*. Laxmi Publications, Ltd., 2008.

[18] W. Kent, *Data and Reality: A Timeless Perspective on Perceiving and Managing Information*. Technics publications, 2012.

[19] G. Kotonya and I. Sommerville, *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[20] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[21] P. Lencioni, *The advantage: Why organizational health trumps everything else in business*. John Wiley & Sons, 2012.

[22] J. Lewis and M. Fowler. (2014) Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html

[23] D. MIT Sloan. (2014) Strategy, not technology, drives digital transformation. [Online]. Available: https://www2.deloitte.com/insights/us/en/topics/digital-transformation/digital-transformation-strategy-digitally-mature

[24] S. Moore. (2016) 7 options to modernize legacy systems. Gartner. [Online]. Available: https://www.gartner.com/smarterwithgartner/7-options-to-modernize-legacy-systems/

[25] S. Newman, *Building microservices: designing fine-grained systems*. "O'Reilly Media, Inc.", 2015.

[26] T. Reenskaug, "The Model-View-Controller (I) Its Past and Present," in *JavaONE*, 2003.

[27] E. Ries, *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books, 2011.

[28] S. Roychoudhury. (2015) Legacy enterprise systems modernization: Five ways of responding to market forces. Cognizant. [Online]. Available: https://www.cognizant.com/whitepapers/legacy-enterprise-systems-modernization-five-ways-of-\responding-to-market-forces-codex1377.pdf

[29] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[30] S. Sinek. (2009, September) How great leaders inspire action. [Online]. Available: https://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action

[31] S. Toyoda. (1930) The 5 whys. [Online]. Available: https://en.wikipedia.org/wiki/5_Whys

[32] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[33] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

A. Change Log

This is the first version so no change log. Yeww!

B. Codebots Field Trials

The codebots field trails are an experiment we conducted to test the following hypothesis: *We predict a team with a Codebot will provide more value to a software development project than a team without a Codebot in the same amount of time.* The way the experiment worked is shown in Figure B.1. In the first week we had a planning session to work on the scope of the project. We then had one of our squads work on the project for a time-boxed week to see how much of the project we could complete. The final step was the review where we would present the results to the participant and ask them to do a survey and provide feedback on the outcome.

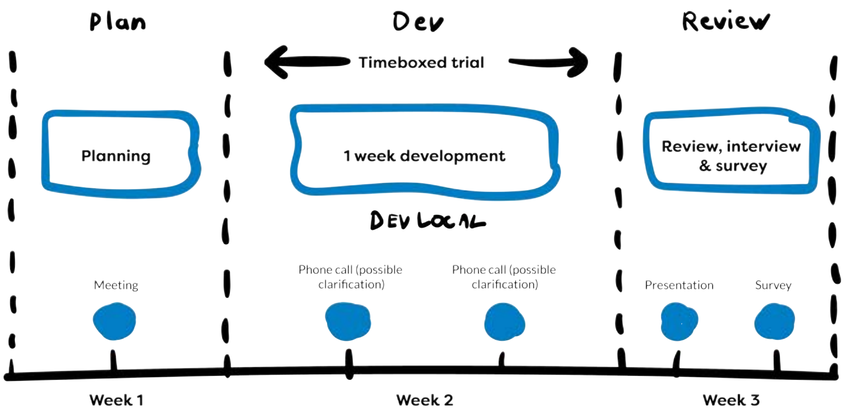


Figure B.1: Time Box

We invited a number of companies to the trial and had six participant companies with one field trial project per company. The companies ranged in size from 40 to 5600 employees and some are listed on the stock exchange. There were multiple people involved from each company, making a total of 21 people that filled out the survey, with an average experience in software projects of 14.83 years. Over all six experiments, the codebots wrote a total of 143,453 (93%) lines of code and the humans wrote 10,864 (7%). Figure B.2 shows the code being developed over the five days.

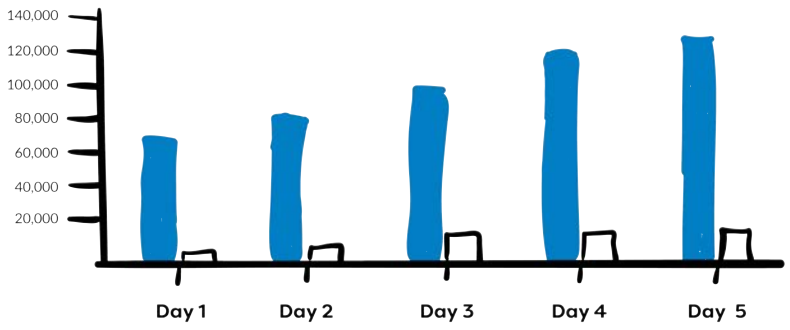


Figure B.2: Bot vs human written code

The other part of the survey was to ask the 21 people how fast a team of comparable size would have taken them internally in the organisation to complete the same amount of work. For the one week's work, the average time it would have taken their team to complete was 8.3 weeks. With 55% saying that the codebots were faster and 45% said much faster (the highest result). The average experience rating was 9.23 out of 10. Of the behaviours listed in Appendix C, the most popular behaviours we used were CRUD, forms, workflows, dashboards, mobile apps, and developer API.

Estimated time without a codebot	Codebots vs regular development speed	Average experience rating out of 10
8.3 weeks	Much faster (45%) Faster (55%) Similar (0%)	9.23

Figure B.3: Statistics

Without divulging the companies involved, here are a couple of project descriptions to understand what the projects involved. For one project, the codebots forms behaviour was linked to a chat interface. That interface had an API linking to the participants Customer Relationship Management (CRM) in order to verify a users identity. Upon verification, the user could request information relating to their account. This was wrapped in an iframe ready to be deployed on any web page. For another project, the wizard and geolocation behaviours were used to create an application that performs business-specific calculations. That application was deployed to iOS, Android, mobile web and as a progressive web app. The data was persisted and historical data could be exported as a CSV.

We will be conducting future rounds of the Codebots Field Trials. If you are interested in participating, please contact BB8 at support@codebots.com. These are the droids you are looking for ...

C. Behaviours

Behaviours are cool things that a codebot can do. In Section 4.1, we discussed the state and behaviour of a software application. *The state of an application is the data and the behaviour is what we do with it*. So, we build our codebots with behaviours that solve common business scenarios. In the table below, we have listed in alphabetical order some of the behaviours that our codebots have implemented. It is important to note that not all the codebots have implemented all the same behaviours, but there are some behaviours that are implemented as they are the most common: CRUD, Dashboards, Developer API, Forms, Import / Export, Timelines, and Workflows.

	Board	Use a kanban-like board where you can visually depict work at various stages of a process.
	Calendar	Display a calendar to be used for appointments, scheduling resources, or booking services
	Chat	Allow direct, group, channel messaging between users.
	CRUD	The ability to Create, Read, Update and Delete your data.
	Dashboards	Display a snapshot of your data with drill-down and segmentation capability.
	Developer API	Use OpenAPI swagger definitions controlled through the security diagram.

	Online / Offline	Store data locally on a mobile-app and sync when the device reconnects to the internet.
$	Financials	Track credits and debit, keep a comprehensive ledger.
	Forms	Create documents, forms and surveys with skip/show logic abilities.
	Gamification	Gamify your application for higher engagement.
	Geolocations & positions	Record geographical position, track and filter data.
	Import / Export	Import and export data from your application via CSV.
	Legacy Database Migration	Transfer your data from a legacy database to a new application.
	Mobile-apps	Build mobile-apps for iOS and Android alongside your web application.

	Payment	Use some preconfigured payment gateways or develop your own.
	Reminders & Notifications	Send push notifications to a mobile-app or other feed like a SMS. Set reminders on timelines to send notifications.
	SaaS	Deploy the application for SaaS using a multi-tenant architecture.
	Shared Library	Share data across multiple organisations or an entire industry if you are using the SaaS behaviour.
	Social Integration	Push content to your social media channels. Display a feed of your previous activities.
	Timelines	Show data on a timeline to reflect history and allow forward planning.
	Workflows	Create a configurable workflow to allow entities to progress through a series of steps.

D. Agile Manifesto

The Agile manifesto can be found at `http//agilemanifesto.org`. The text from the website is shown below.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

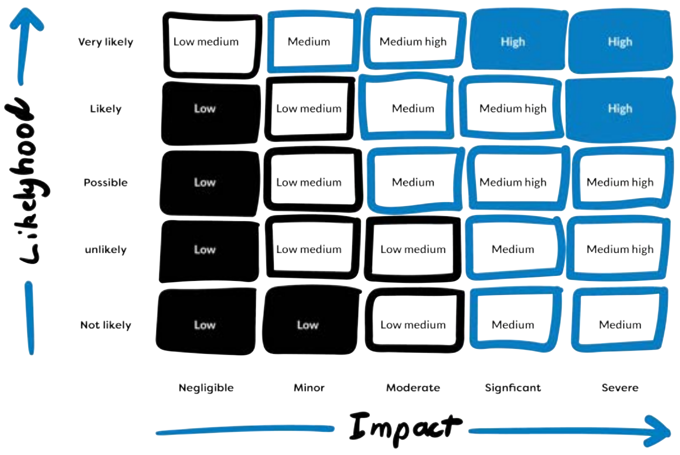
The twelve principles are:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

E. Risk Register

In this appendix you will find the top risks for a project. These risks have been identified from both our own experience and the experience of external projects done by our partners. The risks will change depending on the context of a specific project. For each risk we have included the threat for both Agile and Waterfall style projects. In Figure E.1, is the axis and values we use for likelihood and impact.


Likelyhood
Impact

	Negligible	Minor	Moderate	Signficant	Severe
Very likely	Low medium	Medium	Medium high	High	High
Likely	Low	Low medium	Medium	Medium high	High
Possible	Low	Low medium	Medium	Medium high	Medium high
unlikely	Low	Low medium	Low medium	Medium	Medium high
Not likely	Low	Low	Low medium	Medium	Medium

Figure E.1: Risk matrix

#1 Inaccurate Estimates

Description:	The length of the project, milestone or iteration is under estimated by the project group.
Likelihood:	Very Likely
Impact:	Severe
Threat (Agile):	Medium
Threat (Waterfall):	High
Mitigation Strategy:	• Overestimate and overdeliver • Elaborate only the Work that has immediate priority • Tech Spikes • The Estimation Process • Allocation Factor • The Cone of Uncertainty • Success Sliders
Measure:	• Burn-down chart • Iteration Report • Services Agreement • Scope Document

#2 Scope Variations

Description:	The scope of a iteration is changes after a timeframe has been agreed.
Likelihood:	Very Likely
Impact:	Severe
Threat (Agile):	Low
Threat (Waterfall):	High
Mitigation Strategy:	• Short manageable Iterations • Set realistic goals and manage expectations • Elaboration of only prioritised work
Measure:	• Variation Metric (% scope changed)

#3 End-user Engagement

Description:	Users resistance to change and conflict between users.
Likelihood:	Likely
Impact:	Severe
Threat (Agile):	Medium
Threat (Waterfall):	High
Mitigation Strategy:	• User testing and surveys • Focus groups • Frequent Releases • Beta Testers
Measure:	• Traffic Volume

#4 Stakeholder Expectations

Description:	As time progresses, all stakeholder expectations must be managed.
Likelihood:	Likely
Impact:	Significant
Threat (Agile):	Medium High
Threat (Waterfall):	Medium High
Mitigation Strategy:	• Effective communication • Approval and acknowledge from the partner • The Way of Working
Measure:	• Meeting Attendance • Response times

#5 Poor Quality

Description:	The quality of what is delivered does not match up with stakeholder expectations.
Likelihood:	Likely
Impact:	Significant
Threat (Agile):	Medium
Threat (Waterfall):	High
Mitigation Strategy:	• User Acceptance Criteria • Dedicated Product Manager • The Way of Working
Measure:	• Traceability matrix

#6 Poor Productivity

Description:	The project group is falling behind on the planned time frames.
Likelihood:	Unlikely
Impact:	Significant
Threat (Agile):	Medium
Threat (Waterfall):	Medium High
Mitigation Strategy:	• People culture • Set achievable timeframes • Direct and Constant Product Manager Collaboration • Working at a sustainable pace • The Way of Working
Measure:	• Burn-down chart • Iteration Report

#7 Inadequate Risk Management

Description:	The project specific risks are not managed by the stakeholders.
Likelihood:	Possible
Impact:	Moderate
Threat (Agile):	Medium
Threat (Waterfall):	Medium
Mitigation Strategy:	• Risks analysis included in start and end of Iteration meetings • Include risk analysis into learnings
Measure:	• Including Risk in estimations • Risk Register on Estimations and in the backlog

#8 Low Partner Engagement

Description:	The response time from the partner is slow and impedes agreed timeframes.
Likelihood:	Possible
Impact:	Significant
Threat (Agile):	Medium High
Threat (Waterfall):	Low
Mitigation Strategy:	• Clear agreements • Effective communication around timeframes • Effective selection of Delivery and project goals/priorities
Measure:	• UAT period

#9 Inadequate Human Resources

Description:	A stakeholder must leave the project unexpectedly.
Likelihood:	Unlikely
Impact:	Moderate
Threat (Agile):	Low
Threat (Waterfall):	Medium
Mitigation Strategy:	• Up-to-date documentation • On-boarding with learning guide • The invoice schedule and team utilisation monitoring
Measure:	• Available bench

#10 Lack of Ownership

Description:	It must be clear who is responsible for what and when it is to be delivered.
Likelihood:	Unlikely
Impact:	Moderate
Threat (Agile):	Low Medium
Threat (Waterfall):	Low Medium
Mitigation Strategy:	• Set responsibilities for stakeholders • Clear communication channels • User Acceptance Criteria
Measure:	• Meeting notes and action items

Tackle legacy systems head-on with *Bots that Code: The Continuous Modernisation Playbook*

Continuous modernisation is a powerful new approach to transform complex software ecosystems. *Bots that Code* **is for decision and strategy makers who want to wrestle back control, gain clarity and drive successful digital transformation projects. Whether you're a CEO, CTO or a member of a software team,** *Bots the Code* **will fundamentally shift your perception of what's possible. The playbook includes strategies, tactics and activities that address both the technical and cultural challenges of a software project. When a software/people fit is found, momentum is gained, business agility increases and your organisation will be in a more powerful position to face the future.**

Eban Escott is CEO and co-founder of Codebots with over 20 years experience as an IT professional. Dr. Escott received his Ph.D. from The University of Queensland and this research underpins the Codebots platform.

Indi Tansey is Chief Community Officer and co-founder of Codebots with 10 years experience as a creative strategist. She has launched two tech startups and builds tribes of people around trailblazing ideas.

ISBN 978-0-6486571-0-1

9 780648 657101 >

CODEBOTS

Version hydrogen

www.codebots.com