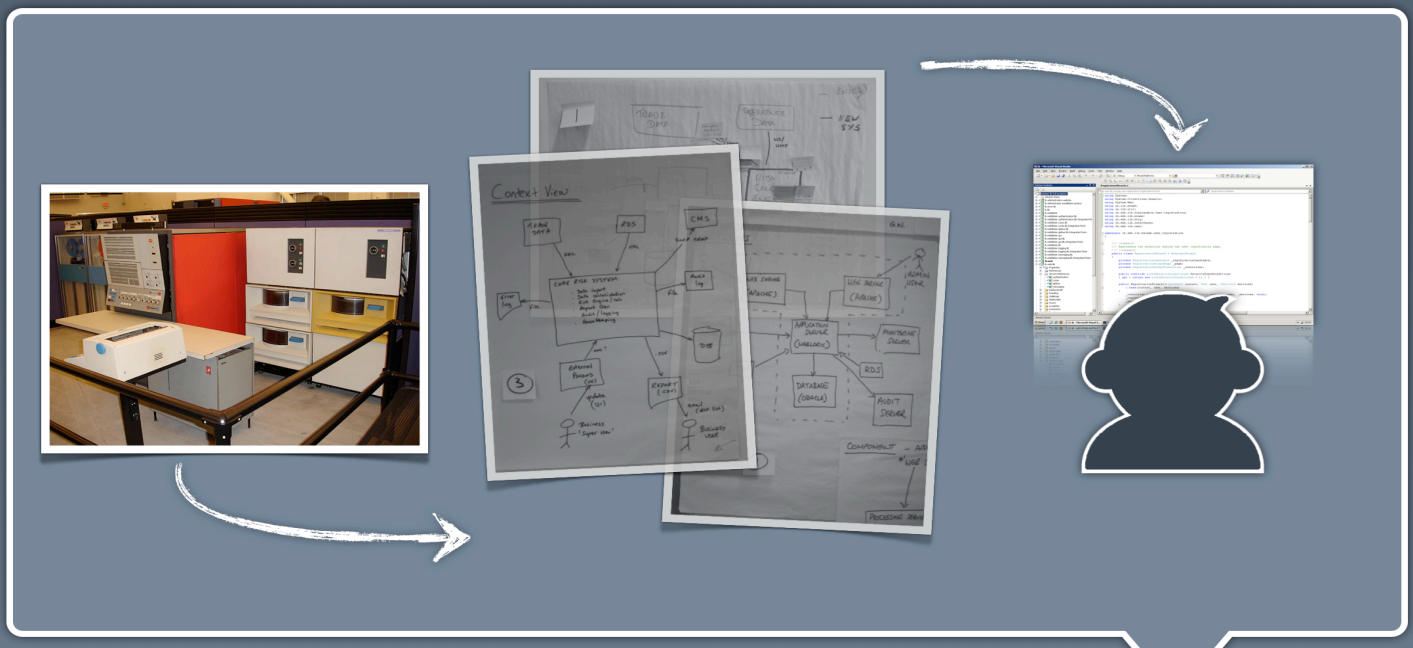
coding {the} architecture

Working with Legacy Systems

A practical guide to the real systems we inherit and maintain.



Issues, strategies, solutions and leaving a good legacy

Robert Annett

Working with Legacy Systems

A Practical Guide to the Systems we Inherit and Maintain

Robert Annett

This book is for sale at http://leanpub.com/WorkingWithLegacySystems

This version was published on 2015-04-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Robert Annett by spreading the word about this book on Twitter!

The suggested hashtag for this book is #WWLegacySystems.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#WWLegacySystems

Hayley, Oscar and Chloe

Contents

Preface

There comes a point in everyone's I.T. career when they become responsible for a legacy system. This is inevitable and I call it the 'Penelope Principle'. Like the Peter Principle (where people are promoted to their level of incompetence) and the Dilbert Principle (incompetent workers are promoted to where they can do least damage - management) this should be accepted and worked with rather than being fought against. It is stated thus:

> "All I.T. workers will be promoted into a position where they become responsible for a Legacy System"[1]

Hopefully, you are nodding your head vigorously at this point (having bought this book this is highly likely) and the purpose of this book is to help you deal with the situation you have found yourself in.

Why is this the case? In Chapter One, I will spend a short while defining exactly what we mean by a 'legacy system', but it comes down to success and longevity. The commercial I.T. revolution started in the 1970s when vast numbers of manual processes and physical records were placed into mainframe systems. Subsequently, these systems have not only been improved but entire new industries created. There are vast benefits to having information systems in electronic form and this effect was magnified by the internet revolution of the 1990s. Some systems don't add value and are scrapped but most do and are therefore used until it becomes cost effective to replace them.

This means there are a LOT of I.T. systems out there involved with every aspect and function of society. This has been happening for almost 50 years so the number of 'old' systems outnumbers the 'new' systems many times over. In the same way that it's impossible to exist without being effected by an I.T. system, it's impossible to avoid legacy ones. Even if you have the world's largest group of developers and an infinite budget in a brand new organisation, you'll still have to integrate with legacy systems and eventually your own green-field projects will become legacy.

If you have a position of responsibility within an organisation, you will have to deal with legacy systems.

However, the I.T. industry is obsessed with new technologies and new projects. University courses, books, magazines and conferences focus on what is new and assume you always start with a clean slate. This isn't what actually occurs in the real world and I hope what follows fills some of this gap. This book is *NOT* intended to present a formal methodology but is aimed at all the Penelope's out there who need a guidebook to help them with their first legacy system.

Robert Annett (robert.annett@codingthearchitecture.com)

[1] The caveat is "unless the worker is so useless they can't hold down a job and keep getting fired before this occurs". This, of course, does not apply to any reader sensible enough to buy this book. (And yes, I do know this is the 'No True Scotsman' fallacy.)

Part 1 - Definition, Issues and Strategy

What is Legacy?

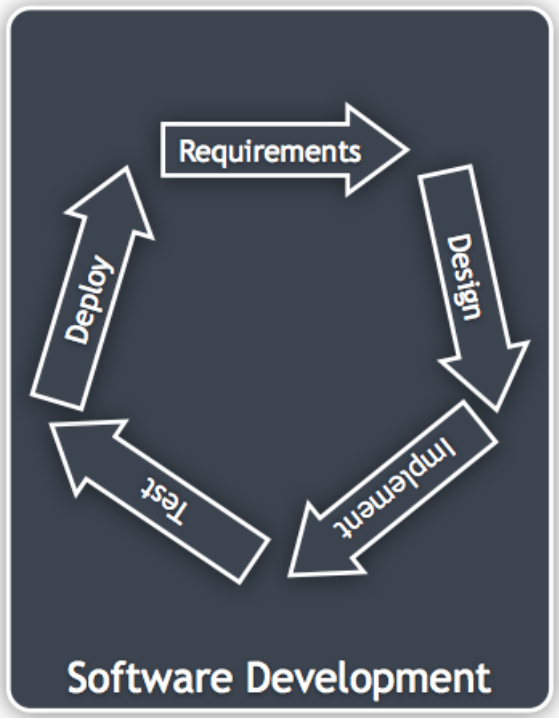
For most people, in most situations, being passed a legacy is a good thing. Perhaps their long lost uncle has left them a stately home filled with antiques. Or a legacy might be a valuable art collection given to a museum or the work of work of a famous author. However, in Information Technology, it is a dirty word filled with innuendo and often used as an insult. This is a strange situation and we should define exactly what we mean by 'Legacy' before addressing how to deal with it - and even if we should deal with it!

When we refer to a System being 'Legacy' what we're really saying is that the system is built in a way that differs from how we'd choose to do so today. Legacy systems may have been written well, using the best technologies and tools available at the time, but those technologies are now out-of-date. A system written in 2001 using Java 1.2 and Oracle 8i may have made perfect sense but if you wrote it now you'd at least use the latest versions available or even something different entirely (Scala, mongoDB, etc. didn't exist then).

An IT centric view of the world

Like most developers, the centre of my work life is the software development process. This might involve a business analysis phase before and some support afterward, but this is a relatively small percentage of my time.

The 'software development centric' view of the world is below (this is the 'SDLC diagram' and your own process may differ slightly). There are two parts that interact with the outside world - requirements and deploy (which may be lightweight and frequent) and the external interactions are wrapped in that.

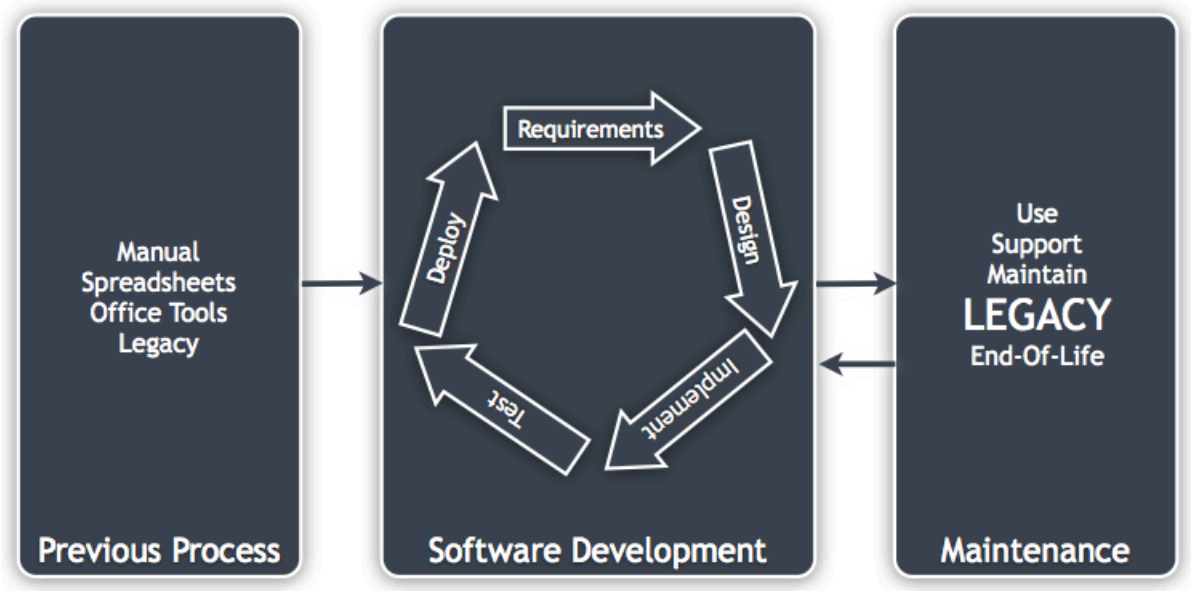


An example software development lifecycle

For some businesses (e.g. public facing web sites) the software IS the business and there is no significant period before or after the software development - it occurs constantly. These businesses are very interesting to us developers for this very reason (and therefore the projects are often discussed at conferences). These are actually the exception. For the majority of organisations, their individual IT systems perform a function that constitutes only a small part of what the organisation does.

Systems Development in Context

Therefore, users of our software view the world differently. The software development phase is a very small part of their business processes, life cycle and lifespan. They view the world a little more like this:



The lifecycle in context

Most of the processes they execute will originally have been done 'manually' (I include generic software tools such as spreadsheets) and may have been done this way from 6 months to 100 years ago. (If you think I'm exaggerating with '100 years' then you should speak to someone who's worked with an old insurance company!)

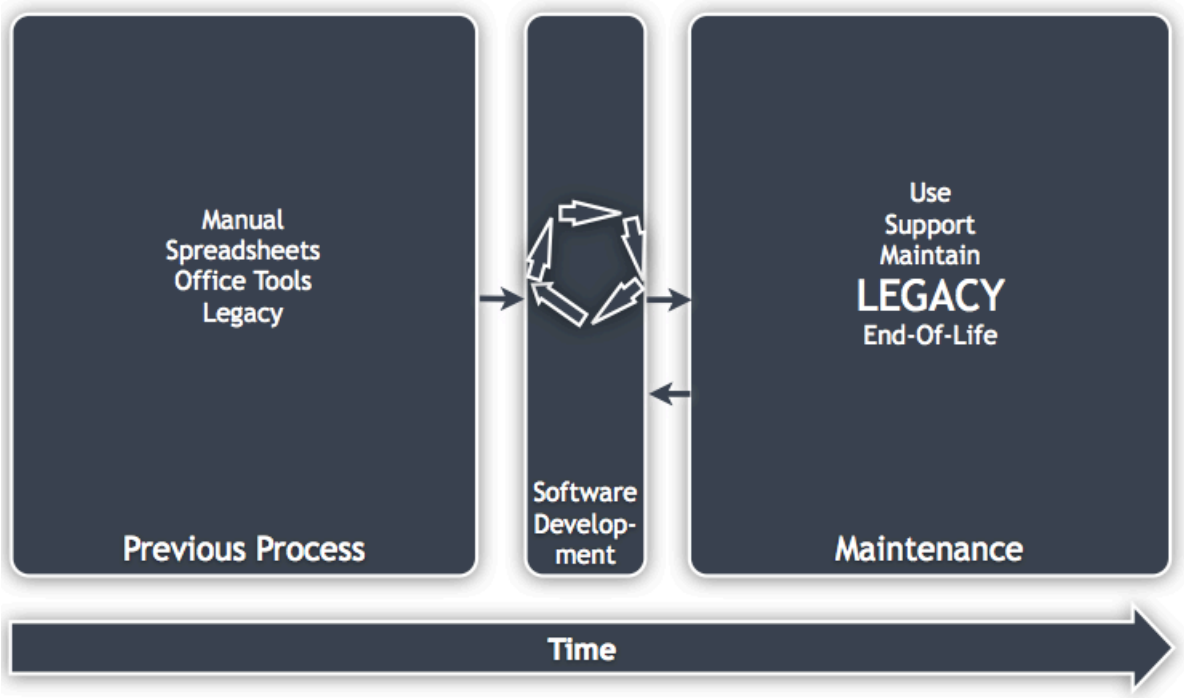
Eventually, it will be decided that it is cost effective to develop bespoke software (or spend a lot of effort configuring BPM/CRM software, etc.). This process may be iterative and deliver value quickly, but will be mostly complete after a relatively short period of time (compared to the organisation's lifespan).

It is now fully live and a software team will consider it to now be in its maintenance phase. From the organisation's point-of-view, this is where the real value is extracted from the system. Very little money is being spent on it but it is being used and is either helping to generate revenue or increasing the efficiency of a process.

As time goes on there will be a decreasing number of changes made to it. Eventually it will be considered 'Legacy'. Ultimately, the system will reach end-of-life and be replaced when it is no longer fit for its purpose.

Systems Development Scaled with Time

If we were going to scale the diagram to indicate time, then it might look like this:



The lifecycle in context and scaled with time

These are typical life cycle times that I've experienced and yours may differ - this will depend largely on the industries you work within.

Therefore 'legacy' should be viewed as a phase in the life cycle of the *process*. Note that I have a small arrow going back to Software Development from the maintenance/legacy phase as it may have upgrades or additions made many years after being deployed.

Line-of-Business Systems vs. Others

It was pointed out to me[a] that the life cycle I've described best fits line-of-business software development and other types of systems that may vary greatly (for example the control systems for embedded devices). This is true and reflects my personal experience but the statements on differing perspectives still hold.

[a] Thank-you to Robert Smallshire

I wrote a blog post covering much of this and someone commented that they refer to 'legacy' systems as 'mature' as this has a more positive connotation. I think it also accurately reflects legacy as being a stage of a life cycle rather than any particular indication of quality.

A system maintained by professional programmers would be periodically upgraded and migrated to new technology when required, so it may never be mature/legacy. However, this is rarely a decision for the programmers - it's a management decision. To be fair to management, does it really

make sense to spend large amounts of money continually upgrading a system if it works? If the programmers and systems team did a good job then no-one might need to make changes for many years. This indicates a high return-on-investment and a good quality system. A Legacy!

Of course, when you do have to make changes or additions you have a very different set of issues compared to modifying a system that is being continually developed.

Examples of Real Legacy Systems

Giving an example in a publication is dangerous as in five years it might be hopelessly out of date. However, considering the subject matter, I shouldn't make this argument. Take a look at the following

© Dave Ross sourced from wikimedia

Is this a legacy system? Actually it's a museum piece. This is an IBM360 and was once the most popular computer on the planet. However, there are no commercially working examples left and they can only be found in museums.

This shouldn't surprise us given the example time line and arguments from earlier. Systems will be end-of-life'd or modernised eventually and it's very rare for a working system to be this old. So, what are the types of legacy systems you're most likely to come across? This will obviously change with time (and depends on the organisational areas you work within) but is most likely to be:

- Between five and fifteen years old
- Without maintenance for two years
- Using out-of-date APIs or technologies

If it doesn't meet these criteria then you are unlikely to consider it to be legacy. As the year is currently 2014, example systems could include:

- Java 1.2 running under Solaris 8 on an Ultra Sparc III server using Oracle 8i
- C# 1.0, Windows server 2000 using SQL Server 2000
- Objective C on NeXT server [2]
- J++/J#, VB6, FoxPro, Brokat Twister.
- A combination of spreadsheets, macros and shell scripts.

I'll cover common issues in the next chapter but the examples above have different problems even though they are technologies from between 10 and 15 years ago. The first two (java/oracle and .net stack) are very out of date but are all supported technologies. These should be simple to upgrade and support but are likely to be challenging to modify. The next two involve hardware and software that are discontinued. This will be much more challenging. Lastly, we have the very common situation of a system bolted together from 'office tools'.

In this book, I'll be covering these kinds of Legacy systems.

[2] I worked for an investment bank that had made a large investment in systems written in Objective C and running on NeXT servers. These systems worked well, were complex and hard to replace. They ended up being run for so long that they had to start buying hardware spares off of eBay.

Common Issues

In this chapter, I'll be identifying common issues with the type of systems discussed in the previous chapter. Depending on previous experience, these might appear obvious (you've seen it before) or surprising (you'll see it eventually). I would suggest that when you take on a new legacy project you go through these and see what applies. Being aware of and expecting these issues will help you deal with them. I have included an 'Issues Checklist' in the appendix to help with this.

No Documentation

When a system has been running for a long time, it is not unusual for supporting artefacts to be lost. The users of the system will notice if it stops working and will complain, but will anyone notice if items, such as documentation, are lost? This can happen for a number of reasons:

- Support systems may be retired and information removed. This can happen when there is no obvious link from the documentation to the system it concerns.
- Documentation still exists but no one knows where it is or if it has been moved and the links/locations are not valid. This is a common problem when migrating document management systems.
- Information is in an uncommon or unknown format. It's amazing how many files end in '.doc'.
- Information was only ever stored locally (and the machines have been wiped) or not stored at all.

The agile manifesto says we should favour "working software over comprehensive documentation" but it does NOT say we should not document anything. In particular, there are various types of meta-data about our system that can make supporting it very difficult if missing. For example:

- Usernames and passwords. The direct users of the system and its administrators may know their access details but what about the system's sub-components? What about administrator access for the database or directory server? If these are lost then performing maintenance actions can be incredibly difficult.
- Release instructions. Maybe you have the source code but do you know how to build and release? Data is more likely to be kept than a running, unused service so your build and deploy server probably won't be present after ten years. Was your build configuration in the source control system?
- Last release branch details. Which one of the multiple code branches or labels was actually released?

- Communication protocols. This is one of my personal bug-bears. Many systems have been designed over the last ten years to communicate with each other via xml messaging. Whereas developers used to document binary message formats, they've often not done so with xml as it's deemed to be 'human readable' just because it's text based. In reality, xml data blocks are rarely obvious and they have so many optional elements that they are very difficult to reverse engineer. This is especially true if you only control one side of the communications, e.g. receiving a message from an external source.
- Licenses and other legal agreements. I'm going to talk a little more about licenses later, but can you track down the legal information about how you are allowed to use the third party software and hardware elements in your system? They may be more restrictive than you think.
- Users, including external systems. Do you know who your users are? What external systems rely on the one you control?

It is always worth tracking down and cataloguing all relevant documentation as a first step to maintaining a legacy system.

Lost Knowledge

There is an overlap between lost knowledge and lack of documentation. You could argue that no knowledge would be lost if everything was documented but this is unrealistic and incredibly time consuming. It's likely the reasoning behind design decisions have been lost. You have the artefact (the system) but not the how and the why as to its creation.

Anecdote Alert!

Many years ago, I worked on a system that, once a day, collected some customer information and emailed it to some users. As part of the process, a file was created (called client-credit.tmp in the /tmp folder on a unix machine). The next day, this file was overwritten. When we began upgrading it, we moved the process into memory so no 'temporary files' were created. I received a very angry phone-call a few days after go-live from someone whose system was now broken. It turns out that another system (in a totally different part of the organisation) was FTP'ing onto the server, copying client-credit.tmp from /tmp and using it.

At some point, there must have been a discussion between the two departments about giving access to this information and how to do it. However, once this hack was implemented (maybe this was not supposed to be the permanent solution) it was forgotten and those responsible moved to different jobs. It was interesting that no-one from either department knew this was happening.

These types of lost knowledge can create some nasty traps for you to fall into. Some of the techniques and processes discussed in later chapters should help you identify and avoid them.

Hidden Knowledge

Also referred to as the 'Special Voodoo'. These are useful but obscure pieces of information which may either work around a bug or initiate a non-obvious function. Although not 'lost' it is also not documented and deliberately hidden by an individual or small group of operators. Why would anyone do this? The most common answer is 'job security'. If you're the only person who knows how to bring up the system when it freezes or create a certain report then it makes you much harder to replace.

This is very, very common in legacy systems and can mean that the very people you need to help you support and improve the system might be actively working against you. I'm going to be covering this a little more in the later section on politics.

Anecdote Alert!

I once worked for an ISV (Independent Software Vendor) that had written a complicated piece of analysis software. This had started as a simple tool and had grown organically over ten years and was now a system used by large organisations. There was an operator for this software at one site who would 'work from home' one day a month in order to complete a monthly report - which he presented when he returned the next day. His colleagues (and boss) thought that he was working all day on the report but, in reality, he actually spent it playing golf. The report only took 10 minutes to generate by exporting the data from an obscure part of the system and then loading it into a spreadsheet template.

Unused Functionality

In a large system that performs complex functions, there is a good chance that there are huge chunks of functionality that are not used. This tends to be either due to an over-zealous development team that added features that have never been used (over engineering) or features that are now unneeded. Often parts of the system are replicated elsewhere in newer systems and we can think of our legacy system as being partly replaced.

Why do I describe this as a problem? Can't we just ignore these? If you don't know whether part of a system needs to be supported or maintained you might waste a long time trying to migrate something that is not used. However, if you decide that part of the system isn't required, and then do not support or maintain it, you might later discover that it was just used very infrequently.

The owners of these rarely used features might be difficult to track down, making it hard to get a sign-off for changes.

Some unused features might still be required even if you never use them or ever intend to use them. This is actually very common in software used in a regulated or safety critical environment. For example, having the ability to delete records for security reasons (privacy regulation) or features used in disaster scenarios. You shouldn't turn off the control systems dealing with the meltdown of a nuclear power station just because "it hasn't been used in twenty years". Unfortunately there are examples of safety system being removed just in this way - it's important you know this.

Even if your system doesn't have a potential catastrophe you should consider whether the Business Continuity Plans and High Availability systems still work.

Is this used? ©Ellin Beltz sourced from wikimedia

Is this tractor used? It's old but not abandoned. It might start but is it used to do any work? Perhaps it's the only tractor in the farm that has a certain attachment to drive an essential, but rarely used, piece of equipment? Importantly, if you're the mechanic on the farm do you spend a huge amount of time keeping this working? Can you even find anyone to ask about this?

No Coherent Design/Inconsistent Implementation

This can be due to an active decision to avoid top-down design but is more often due to a system growing in an organic way.

Many successful, large systems don't start as large systems. They start as a useful tool doing a difficult but specific task. If they do the job well, then they'll have features and options added. Over a long period of time it can morph into a large and complex system with no coherent design. Common indicators of this are:

- Business logic in the incorrect place such as in a client rather than service. Perhaps this started as a stand-alone application which morphed into a multi-user application.
- Multiple tools to doing the same job - for example different xml parsers being used in different parts of the system or several ways of sending an email report. This is often due to a new developer bolting on a feature using the tools they know without looking at the rest of the system.
- Inappropriate use of frameworks. For example, using threads against advice in an application server or trying to get a request-response style service to schedule tasks. This is normally due to adding a feature that was never originally envisioned and trying to do it in the current framework.
- Many layers and mapping objects. New features may not fit into the current 'design' and require a lot of supporting code to work with what was there.
- Sudden changes in code style - from pattern use to formatting. Where different developers add features at different points.

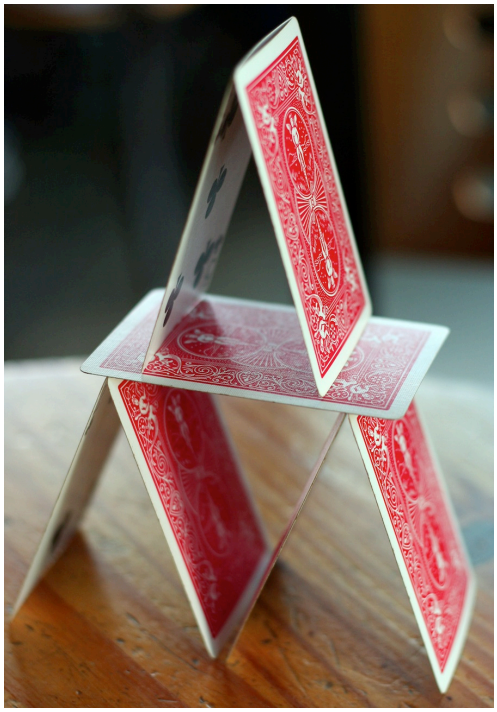
Some of these may be bad (time for a refactor) but for others, it may just be inconsistent. Why does this matter?

Lack of coherence in design and consistency in implementation makes a system much harder to modify. It's like driving in a different country - the road systems may make perfect sense but they take a while for a foreigner to get used to - and if you move countries every week, you'll never be a good driver again. Similarly, it does take a developer a while to get used to patterns and tools and they will be much less productive in an incoherent system.

It's much harder to predict the non-functional behaviour in such a system. If each part of the system works differently, then performance tuning becomes hugely complex.

Fragility (vs. Stability)

The first thing to note is that I've used the term "fragility" rather than "instability". The difference is subtle, but important. An unstable system will stop working in a frequent and unpredictable way (if it was predicable then you'd have specific bugs). Whereas a fragile system will work as required but will crash horribly when a change is made.



It's stable but fragile. Don't open a window.

Fragility can be worse than instability from a maintainer's perspective. If the users consider a system to be fragile, it can be difficult to get permission to modify it. Sometimes this fragility may be perceived rather than actual and this is common when the system is old and no longer understood - the fear is that making any changes will stop it from working and no one can repair it.

In contrast, sometimes an unstable system can work in your favour as you can make mistakes and no one will notice - "whoops, it crashed again!"

Later in the book, I'll be covering some of the tools and techniques for making changes to fragile systems.

Tight Coupling

In a tightly coupled system, it can be very difficult to pull apart the different components of a system. This makes it hard to upgrade in a piecemeal fashion leaving you with a stressful 'big bang' approach to upgrade and maintenance. It's also very difficult to performance tune problematic functions.

Loose Coupling

What? Surely loose coupling is a good thing? However, what about the anecdote I told earlier about the temporary file which was FTP'd by another service? Although this is a terrible design, isn't it an example of loose coupling?

With a tightly coupled system it's usually obvious what the dependencies are and nothing runs without them being in a specific state. With a loosely coupled system (particularly one where services depend on each other and communicate asynchronously via a bus) this may not be so obvious. This is particularly problematic when there is a lot of unused and rarely used functionality (see earlier issues) as you don't know how or when to trigger potentially important events.

Zombie Technologies

The reason this is called 'Zombie' rather than 'Dead' is that when I actually looked into some example technologies I find out that almost no technology actually dies. There is always some hobbyist somewhere insisting that it's still alive, therefore, I'd count the following as zombie (dead in reality) technologies:

- Technologies where the company that created and supported them no longer exists
- Technologies no longer officially supported by their creators (including open-source projects that have not been touched in years)
- Technologies not compatible with the latest versions
- Technologies where it is not possible or practical to find engineers that have any knowledge of them
- Technologies where important parts have been lost (such as the source-code)

I'm sure I'll get complaints about some of the above definitions but these are all situations that make a system much harder to upgrade or maintain. Trying to solve a bug or modify the deployment for these can be difficult and modifying functionality might be impossible.

We should also remember that fashions and the way technologies are used can change a lot over time. Well written Java code that used the best practices from 2001 will look completely different from the code a Java developer would write today.

Licensing

All of the components in a system, both software and hardware, will be covered by some kind of license. This is something very often forgotten by technologists when inheriting a legacy system. The licenses can say almost anything and might put very restrictive conditions on what you do with the components. This applies to everything through the entire stack and potentially even some of the data in the system if it was supplied by a third party with restrictions on its use.

Here are some questions you should ask about the system and licenses.

1. Can you make changes?
2. Does it stop you using virtualisation?

3. Are you not allowed to run the software on certain hardware or the hardware only run certain types of software?
4. Is 'reverse engineering' prohibited and will this affect your ability to instrument it?
5. Can you modify any source you might have?
6. Can you even find the licenses?
7. Is any of the data owned by a third party?

The cost implications of not understanding the licensing can be high. In the January 2013 issue of *Computing Magazine*, some of the implications of licensing and virtualisation were discussed and the following quote given:

Hidden Licensing Costs

"If a company has SQLServer on VMware, licensed on a per core basis that can be dynamically scheduled across hosts - all machines need licenses." - Sean Robinson, License Dashboard quoted in Computing.

As well as the problems of restricted use, the licenses are also a 'sunk cost' (see below). If the organisation has paid a large amount of money for a particular product then it can be politically very difficult to change, even if the alternative has a lower overall cost.

Sunk Costs

Sunk Costs are retrospective (past) costs that have already been incurred and cannot be recovered. It is an interesting area of behavioural economic research. In theory a sunk cost should *not* affect future decisions - it is already spent and can't be recovered. In practice, people find it very difficult to 'write-off' previous expenditure even if this makes economic sense.

Regulation

Regulation is very similar to licensing in that it also concerns external rules imposed on a system that are outside the normal functional and non-functional requirements. Although licenses stay the same when you want to change the system (and maybe being in violation) the difference with regulation is that it tends to change and force functional changes on you.

Shortly before writing this book, there have been changes to many websites in the UK. A huge number have started informing users that they store cookies and requesting permission from users to allow it. This is due to the Privacy and Electronic Communication Regulations (PECR) Act 2011

which is the UK's implementation of the European Union's ePrivacy laws. It's often referred to as "The Cookie Law".

Of course not all websites have implemented this and the ones that have are the high volume websites with a permanent development team. Many smaller organisations with legacy websites that aren't actively developed, have not started asking these questions and are arguably not compliant. I'm not a lawyer though, so seek legal advice if you think you have a problem!

It's interesting that so many organisations have taken this regulation so seriously and it's probably because the use of cookies is so easy for an individual outside the organisation to test. We can imagine someone writing a script to find UK based websites that don't comply and then trying to sue them for infringing their privacy rights.

There are many other regulations that affect IT systems from industry specific ones (finance, energy and telecoms are heavily regulated industries) to wide ranging, cross industry ones, such as privacy and data protection. Regulation of IT systems are increasing and systems normally have to comply even if they were written a long time before the regulation came into existence.

Please remember that failure to comply with regulation may be a criminal offence. If you breach a license agreement you might find yourself being sued for a large amount of unpaid fees, but if you fail to comply with regulation (for example money laundering or safety reporting) the consequences can be much worse.

Politics

Technology workers are often very bad at spotting political factors that affect their projects. Here are some questions you should ask yourself about your project (whether legacy, green-field or somewhere in-between).

- Who owns the project?
- Who uses the project?
- Who pays for the project?
- Who gets fired if it fails?
- Whose job is at risk… if successful?

What we should be looking out for are conflicts of interest - so let's look at the last two I've listed.

If a project fails then the sponsors of the project and the implementers of the project will take the blame and their jobs may be at risk. This is to be expected and is one of the reasons we're motivated to do a good job and why we get harassed so much by the project sponsors.

However, we rarely consider who suffers when the project is a success. It's much easier to get funding for a migration or upgrade if there is a defined business benefit. Even easier still if there are increased revenues or costs savings (profit!). When an IT system saves money, it's often because jobs can be 'rationalised' i.e. people get fired.

Anecdote Alert!

Once I was taken to see a client by a salesmen for the software company we worked for. We walked through a room of data entry clerks (most of whom were women) and being a stereotypical salesman, he stopped to flirt several times (he already had a couple of divorces and was working on his third). Once we left the room he said "It's a shame isn't it?". I was a little confused and asked what he meant. His reply was "Well, once we've installed the data importing software you've written, they're all out of a job". I hadn't been out of university for long and was naive to the real world effects (and potential politics) to what we were doing.

Even if jobs aren't made redundant by a system improvement, the user might suffer in other ways. If the tasks become standardised/commoditised then it's much easier to replace a specific user with someone else who is cheaper (outsource the processes) or has a better attitude. Remember the examples of hidden knowledge I gave earlier? An improved system might remove this advantage for the power users.

We have to remember that many people simply don't like change. Technology workers are actually very unusual as we like change. It's often what drew us into the job in the first place. We relish the way we work changing and making ourselves redundant probably holds no fear as we move jobs every 18 months anyway. The users of a legacy system may have been doing the same job in the same way for ten years and the thought of learning something new (and possibly being bad at it) can scare them.

You should also question the costs and benefits. There is a good chance that the cost is incurred by someone different to the beneficiary of the project. This is often the case in larger organisations where the IT department is separate from the business units actually using the software. Different solutions to issues may incur cost to different groups. For example, a re-write of a piece of software may come out of the business unit's budget whereas an 'upgrade' might be viewed as the IT department's cost. Virtualising an old system will probably be a cost to the IT department but the cost increase or savings on business software might fall to the end users. This will cause conflicts of interest and the decisions taken may be based on the political power of the department leaders rather than being the best decision for the organisation as a whole.

Politics can be very destructive and I've seen this as the cause of failure for many projects. From personal experience, I'd say that it's more often the cause of failure than bad technical decisions or poor implementation.

Organisation Constraints and Change are reflected in the System

Beyond the deliberate effects of politics there are more subtle marks that an organisation can leave on the systems that are developed for it. This is described as Conway's Law[3].

[3] http://en.wikipedia.org/wiki/Conway's_law

Conway's Law (from Wikipedia)

Conway's Law is an adage named after computer programmer Melvin Conway, who introduced the idea in 1968; it was first dubbed Conway's Law by participants at the 1968 National Symposium on Modular Programming. It states that "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations". Although sometimes construed as humorous, Conway's law was intended as a valid sociological observation. It is based on the reasoning that in order for two separate software modules to interface correctly, the designers and implementers of each module must communicate with each other. Therefore, the interface structure of a software system will reflect the social structure of the organization(s) that produced it.

Not only are these organisational constraints reflected in the system *when developed* but these constraints will change over time. As people leave, teams merge or split and the barriers that existed may disappear or new ones created. However, the artefacts in the system will remain and build up over time. This can become very obvious in long-lived legacy systems.

This affects software and hardware. Consider a system which two different departments in an organisation are using. They may want separate databases (that only they have access to) and may want their processes/reports run on servers they own (perhaps they had to provide some budget for the system's creation and they want exclusive use of 'their' server). These departments may change dramatically over time but you'll still have two databases and two servers.

Understanding the organisational structures of software and business teams can help to explain which quirks of the system are due to this and which are due to some large, complex subtlety that you're yet to understand[4].

External Processes have evolved to Fit around the System

An IT system is part of a larger process and workflow. It will have been developed to fit into the process to make it more efficient and effective. However, it doesn't just fit into the outside world, it will also affect it. The processes surrounding the system will change and evolve to work with it and its quirks. With a legacy system, these processes will have become embedded in the organisation and can become very difficult to change (remember that people don't like change).

This can lead to the peculiar situation where you have to get a newer, better system to work with old and inefficient processes that were originally imposed by the proceeding system. When gathering requirements for a new system, you should remember this and make sure you are finding out what is *needed* rather than what is *currently done*.

[4] Hat tip to Richard Jaques whom I stole this sentence from.

External Systems have evolved to Fit around the System

The external systems that interact with the legacy system will have also evolved to match its interfaces. This is especially true if the legacy system is quite old and the external systems were created afterwards.

This can lead to the situation where a replacement system has to match old and unwanted interfaces in order to interact with dependent systems - that only have those interfaces to connect to the legacy system that is now being replaced. Ancient interfaces can therefore survive long after the system they were originally designed for have been removed…

Decaying data

I've already described how systems can grow organically, but the data within them can also decay[5]. This is a decrease in the overall quality of the data within the system caused by small errors gradually introduced by the users and external feeder systems. Examples include:

1. Data entry (typing) errors.
2. Copy-and-paste reproduction errors such as missing the last character or extra whitespace.
3. Old data i.e. the details are no longer true such as an address changing.
4. Feeder systems changing formats.
5. Data corruption.
6. Undeleted test data.

When a system is new there are often data quality checking tasks to keep the data accurate but with time these tend to be dropped or forgotten (especially tasks to prune data no longer required). This is another example of something from behavioural economics called "The Tragedy of the Commons".

The Tragedy of the Commons

When many people have joint use of a common resource, they all benefit from using it. The more an individual uses the resource then the more they personally benefit. If they mistreat it or overuse it then it may degrade but the degradation is spread amongst all the users. If an individual spends time improving the resource then any increased benefit will also be spread amongst the whole group. From a purely logical point of view it makes sense for any individual to use it but not spend any time improving the resource. They are acting independently and rationally (according to self-interest) but the resource will degrade over time.

[5]The metaphor isn't perfect but I'm trying to make this interesting!

These errors in the data can (and sometimes without being noticed) accumulate until the entire system becomes unusable.

Now What?

Did you recognise any of those issues from previous projects you've worked on? I've not listed the system being 'bad' as a problem as it might not be perfect (all real systems have issues and bugs) but if it's a legacy system then it must have value or it would just be turned off.

When working on a legacy system, it's worth recording issues and potential issues like those listed above. Consider starting a 'Problems and Issues' document for your legacy system and recording real and potential problems.

What is in the Full Book?

This sample book only contains the first two chapters. The following chapters are in the current, full book but remember that this is an incrementally published book so more chapters will be added.

Reasons to be Cheerful

I spent the previous chapter identifying problems but I want to be positive about the situation we often find ourselves in - and I think that some of the issues with legacy system are due to neglect caused by negativity. So here are a few of the reasons why working on a legacy system should be viewed as a positive.

Strategies

Once you have inherited a legacy system, what do you do with it and how do you approach any issues? Most books on software and system design assume a green-field project where you can build how you want. Legacy systems have constraints on what you can do and how you can do it. This chapter suggests some strategies for managing these systems.

Stakeholders

Whatever strategy and approach you take, it is important to identify and analyse the stakeholders concerned. This chapter covers some common legacy system stakeholders and their concerns and needs.

Architectural Sketches

Many books on software architecture and design assume your projects are ‘green field’ with no legacy. This chapter gives an overview of the C4 architectural sketching method and applies it to a legacy system.

Further Analysis

The analysis suggested so far have been static and high level. This chapter contains a few lower-level analysis areas that may apply to your system. Each of these short sections could justify their own chapters (or even whole books) but I give a few pointers for brevity.

Safely Making Changes

This chapter covers how to make changes to your legacy system in a predictable and safe way. Much of this should be useful even if the system is intended to be kept as functionally unmodified as possible.

Preparation Issues

The previous chapter, on making changes safely, all sounds too good to be true - and of course it is. There are many issues which might cause you to deviate from this plan or require a work around. This chapter describes possible issues and suggestions for solving them.

Stabilisation

I would always suggest performing a stabilisation phase when working with a legacy system - even if you intend replacing it. If the system is stable then you will not be constantly pestered by the user's complaints. This chapter covers some basic actions you can perform to stabilise your system.

Leaving a Good Legacy

So far I've discussed the problems, strategies and techniques to dealing with a legacy system that you inherit. Much of this assumes that there are negative issues you need to deal with. This chapter describes what I believe you need to do to avoid creating these issues when working on a new system.

Appendices

Appendix 1 - Example Legacy Scenario

Very Brief Overview

The salient points:

1. You work for a furniture company
2. You get a promotion!
3. You are now responsible for the warehouse inventory system…
4. No one has touched it for a while.
5. It was written in 2003
6. It's a 3-tier architecture etc etc…
7. It basically works, although people moan about it.
8. What do you do?

Why this example?

Most organisations are NOT primarily focused on technology! One of the reasons I have picked a furniture company is that any IT systems they have will support the main business rather than being the revenue driver. This is true for most IT systems in most organisations - web companies selling online services are not representative.

You have found yourself responsible for this system and discovered that no one has DEVELOPED (or performed maintenance) programming on this system for a while. However the system is definitely used and performs an important and core (supporting) function within the business - no one gets their furniture if the warehouse is not operating correctly.

How should you respond to the situation you find yourself in and what problems are you likely to face? What are the strategies you should employ and how should you execute them?

It's very difficult to get a good impression of what this system does, who uses it and how they use it from the description above (or any text only description). Hopefully the diagrams and sketching techniques, used in the planning chapters, demonstrate how useful they can be.