SOFTWARE MAPMAKING

Making modern life better – and the end of the era of the geek

Karl Jeffery and Dimitris Lyras

September 2019

CONTENTS

WHY SOFTWARE MAP MAKING

Introduction

We think there is scope for modern life to be better - because there is scope for the software which supports modern life on to be better.

This book suggests a way it could be done.

Our recipe, which we call map making, means spending more time looking at software in an abstracted way – as a plan, map, or model – and spending less time looking at the code.

And keeping this map in the forefront of the software development work, as a plan to be followed rather than a code. In the same way that a builder follows a plan, a musician follows sheet music and an actor reads out the lines provided by somebody else.

You'll be wondering why we think that can make so much difference and if so, why it isn't being done so far.

We are not thinking about all software – we are thinking particularly about software which is used to co-ordinate complex activities, such as running a health service, smaller businesses, personalised education, or development of digital technology itself.

The problem which we think we identify is that so much of the work of developing such software involves focus on the technical details needed to make the thing work. This leads to a certain mindset in the people involved in making such products. This may be the mindset you identify with the word 'geek'.

Technical people are accustomed to leaping into battle with code and their machines - but perhaps less accustomed to strategic thinking at a different level.

This would be the right mindset for hand to hand combat in a war, but not the best for figuring out if you can avoid the conflict happening or resolve it in other ways.

It is not enough to just simply say people should stop looking all the time at the details and look at something in different levels. Somebody needs to figure out ways to do it and share them. This is what we try to do in this book.

You may ask what problem we are trying to solve with this book.

The problem we see is that the people who have work involving complex co-organisations or projects – such as a health service, smaller businesses, personalised education, or the development of digital technology itself - are rarely given the digital tools they need to do their job as well as possible. The same applies to all the individuals working in these organisations.

We are not talking here about retail / consumer software, with enormous markets, which is generally much better built.

We are talking about the millions of smaller software markets where people all work in different ways, and would probably need software made just for them, to get the full benefits.

These people who use software to make decisions which impact our lives in many ways.

If this idea sounds far-fetched, think of how much time people in your organisation spend updating each other with e-mail and meetings – and how much may be possible to do with automated tools, if they were built in exactly the right way.

Supporting co-ordination

Digital technology does specific things for our society. It provides us with entertainment, it stores and serves us with data, it performs complex calculations, and it can help to co-ordinate complex activities.

As of 2019, its capability to do the first three of those is very well evolved. But not the last one.

A well co-ordinated organisation can do much more with its available resources, including people, physical assets, and knowledge. There are plenty of organisations which would appreciate being able to achieve more from what they have.

We think there could be rewards for people who master mapmaking for software, taking ideas from this book.

What a world with better software could look like

Software developed through mapmaking would be much more capable of supporting complex co-ordination which modern companies require. It automates the process of giving people the right information at the right time.

This enables the organisation to make much more effective use of its available resources – including data in the organisation, people’s time, and physical assets.

Software made through map making might be able to achieve results which software made in conventional ways are unable to achieve. Software companies will compete on how well they can make software which models, or meets the close needs, of their customers, rather than how efficient their software development processes are.

It can help make our jobs much more interesting. And it all leads to better organisations. That makes a big difference to all of us, particularly for organisations in sectors such as health, education and justice, where resources are stretched and the demand for sophistication quite high.

Consider how many examples there are in daily life where there are in-optimum outcomes due to poor co-ordination between people in organisations, or people not having the best information that a machine could give them.

This can include anything from delays in airports to people dying unnecessarily in hospitals.

Poor education outcomes, poor provision of justice, high crime levels, and any poor performance of a company.

Here’s another specific example. Many parents worry, with good reason, about the videos their young children watch online, or the communications they have with strangers within their computer games.

It would not be too much of a technical challenge to give the parents a copy of just the list of videos or the text based communication. It would require co-ordination communication between multiple internet companies, which appears to be impossible to achieve, because of the lack of a mapmaking approach makes it too difficult. So we don’t have this.

Or consider the mental health services provided in your district. Mental health problems are everywhere, some more serious than others, and there are professionals with skills to support problems, and many volunteers willing to help. But the dots are not usually joined up well enough to ensure that the people with the biggest need get the most support.

Consider public transport provision. In the UK today, people have a choice of purchasing cars to drive from door to door, paying for inexpensive but rigid rail tickets, or paying for expensive and flexible rail tickets. It is not beyond the capability of software to deliver a system which enables low cost and flexible public transport, if it was designed to work with how people wanted it to work.

Digital technology could remove all the bureaucratic headaches of modern life, when we are dealing with insurance, health, visas and passports, tax, applications, claims, broken home broadband, employment, but it doesn't.

Digital technology could do far more to help fix really difficult social problems - refugees, sociopathic neighbours, violent crime, terrorism, but it doesn't.

Digital technology could help fix climate change by making it easier for us to see options which give us what we need without CO2 going into the atmosphere, but it doesn't.

We could have much more effective systems to manage our finances. For example, if a bank suspects our credit card number of being stolen, it could just increase the suspicion level of transactions rather than block our card entirely. It could allow us to continue to make transactions if they fit our usual pattern of card use.

We could have software which makes it easier to give digital money from one person to another – and interrupted far less often by security concerns from our bank.

We could have better systems to help us book tickets and do other tasks online, where the computer system is better modelled around the sort of things a normal person might want to do, and can help us get there. Rather than today's systems which leave us scratching our heads to work out the computer's logic in order to do what we want.

It would make it much easier to build software which lets people get better feedback on their own decisions. So many professionals - doctors, engineers, clinicians, project managers, marketing people – make decisions and never see the results of them, because it is so hard to do with software – connecting a patient outcome with a decision a doctor made two years before.

We could avoid physical infrastructure at any of our national borders – all the checks, tariffs, and anything else associated with a border could be handled in advance with software. The only infrastructure necessary at a border would be checking that what is crossing the border is the same as has been agreed, which could be achieved perhaps with just a drone or satellite imagery.

A side advantage of better software is that the people working with it could have much more enjoyable working lives – as a result of having much more engaging, absorbing information to work with.

Having millions of people happier and more comfortable would lead to the end of voting for populist politicians, and anxiety driven options.

We think the software market may be moving in this direction.

Real working life is about goals

In real life, that is, everything outside the software world, all of us have specific goals. The role of the digital technology is to help us to achieve them. Our suggested process for making maps for software begins here.

Success in real life doesn’t happen by accident, or as a result of great systems. It happens as a result of people with great skill focussing on making it work, and continuously looking for ways to improve how it works, including fixing problems.

These are the people who deliver good organisations, good rail services, good health services, good schools, good transport systems and everything else.

These people have specific targets or objectives, such as providing a good hospital service, keeping children safe online, delivering post, providing a return on investors' capital, keeping a business profitable.

Achieving these objectives is usually complex. A hospital service means being able to give local residents what they require when they require it. Safety online means letting children roam online quite freely while blocking anything harmful. Delivering post means getting post to the right address in a sensible timescale. Return on investors' capital involves keeping investors informed and maintaining their confidence.

People have a range of resources at their disposal to help achieve their goals, such as, in these examples, hospital beds and doctors, technology monitoring tools, postal vans and staff, places to invest.

People do their jobs with a variety of mental maps about what 'good' means in different resolutions, or what specifically they need to do to achieve it. The existence of these maps is the reason these people are doing these jobs, and not you or me. They take years to develop, including learning from people who spent decades developing their own maps, and working together with people who have their own maps.

The task of building software begins by understanding the mental maps of the people who already make decisions in that domain – and using that as a basis for the map of the software.

Software needs to be orientated around goals

Today’s organisational software is generally not modelled around goals or objectives of the customers or ‘users’ of the software.

More likely, we’ll find software modelled around specific things, or ‘objects’.

As we’ll explain in detail in later chapters, making software around objects made a lot of sense in past decades, when software was much harder to make, less powerful, and generally only used to manage things which could in fact be considered as ‘objects’, such as library books or bank transactions.

But as software gets much deeper and more entrenched in our working lives, it increasingly hits against the problem that working people don't really see much of their working lives as objects. Consider a farmer. Yes, a tractor is an object, but to the farmer it is a means of cultivating the land, providing food and making money.

Software built around objects can be very inflexible, because the core of the software is a structure of object attributes, such as a database.

But in real life, we see that while goals don't change much, what we consider as relevant 'objects', and how they interact, change all the time.

We have different tools and objects today than our ancestors did, but we still want to eat, learn, make money, socialise and get to where are going just like they did. So software geared around goals can be far more sustainable and flexible.

Bear in mind, no company got competitive advantage through better management of its objects, such as its management of its accounts, transactions, inventory or contacts. Companies get competitive advantage because they are better at achieving their goals.

Goals also give a common language between different professionals, far more than objects do. Consider two primary school teachers, who may work in completely different teaching systems in different countries, meeting for the first time.

The key 'objects' of their work, and how they are used, could be completely different, and they would not have a common understanding talking about them. For example, their registration system, daily schedule, physical objects and curriculum.

But it is easy to imagine they might quickly find a common understanding talking about the challenges in meeting their goals, such as how to ensure every child leaves the room inspired on their available educational budget, or how to handle the inevitable class troublemakers.

So making educational software might be a far better business proposition, and far more sustainable, if it was designed around goals rather than objects.

Here's another illustration. If we want to compare two toasters, there are plenty of technical specifications we could use. But it would be far easier, particularly if we are having a conversation about the toaster with someone else, if we had a discussion about how well the toaster could help us meet our goals – of toast?

A bigger example of the problems from our object centred software is arguably the Boeing aircraft crashes in 2018-2019. 189 and then 157 lives were lost in two crashes. The cause was software automatically adjusting the pitch of the aircraft due to calculations from sensor data. But the data was wrong because the sensor was malfunctioning, and the computer did not know.

An aircraft pilot, like all experts, is trained to consider the credibility of all new incoming information, as an important part of the map of how to achieve the central goal of safety. A human pilot would surely not have made a major adjustment to the plane based on new data from a single sensor, without first considering that the sensor may be faulty.

So if the aircraft software had been build more closely around how airline pilots achieve their goals, rather than standard software processes around objects, the crash might have been avoided.

What we can do with better software

Changing the way that software is made would be a big effort. So for the ideas in this book to have any value, it is important to continually emphasise the benefits we could get from software made in a different way.

On the surface, today's society, and software products, feel fine. But looking under the skin, we can see that our working worlds have organised themselves around what software does well, without anybody noticing.

Any task involving databases – such as inventory, logistics, billing, appointments, is handled to a computer and probably has been for a few decades now. We are all used to this and work on the assumption that it will happen.

Consider the service contracts we take out for our house heating system, paying a monthly fee for an annual service and a repair engineer available when we need it.

We are not surprised at all to find that the service company has a database to keep track of the clients and their monthly fees, and which mails out letters, and schedules engineers to do the annual service.

On the day of our annual service visit, the engineer calls us in the morning to remind us of the visit, which will be between 1pm and 6pm.

When we tell the engineer that we need to be out from 3.20 to 3.40pm to collect children from school, we are not surprised to be told that the appointments can only be made in the order generated by the computer system and no further flexibility is possible.

We have been, in effect, trained to accept a rigid but sophisticated computer software at the heart of the service.

Here are some more examples. We will expect our dentist to use software to manage appointments and payments. We don't expect our dentist to have a warning system to say that current thinking among a consensus of dental professionals recommends a different course of action for that specific problem. This is possible with today's technology but much harder to build.

We will expect our road maintenance engineers to work on a schedule but not to be aware of similar work happening by another group of engineers on an adjacent road, with mayhem caused by both tasks happening at the same time. That would need a data management and co-ordination sharing capability which is far harder to build.

We expect our parcels to be delivered by a courier which has a computerised system to manage and schedule deliveries, but has no way to understand the relative urgency of each parcel, other than the delivery service chosen by the sender.

In the retail world, it is normal for computers to do what they can easily do – make automated orders to restock shelves, based on purchase levels and estimates of future requirements. People do the elements which machines don't easily do, such as sales, manual shelf stocking, scanning purchases, product design.

And meanwhile we spend so much of our lives queuing, trying to reach customer service people by phone, e-mailing, co-ordinating in meetings, waiting – all tasks which better software could eliminate.

In short, we have all got accustomed to software which is good at managing objects. Conversely, we have got accustomed to not complaining when we don't have software which could do tasks which are hard to define as objects.

Why you should be a map maker

Until now, the most interesting jobs for people with technical skill have been people who can take a role in big complex software projects, keeping the software on the road. But these tasks should surely be getting easier and easier, as we get more standard platforms and more reliability.

There have been many interesting technical jobs over past decades in creating radically new products. But does it really make sense to keep looking for those? It is hard to think of a breakthrough technology which has happened since 2009, when we already had the iPhone, Uber, Amazon, Facebook, and the hardware beneath it all. Chips are not getting much faster, and our handheld devices can do far more than we can ever want.

But some of the best jobs in society and technology in the coming years might be for the map makers – the people who can figure out how digital technology can help organisations meet their goals – and guide the projects to develop it. People with understanding of the world and the capability to apply computer skills to it.

People who have a high understanding of how things work in a broad way - scientists, philosophers and artists. People with entrepreneurial skills, people with ability to develop new financially viable products – and the capability to apply this to technology.

People who are able to think in abstractions and understand which abstractions are the most important, in a technological environment. People who can see the difference in relevance of a useful status update – instantly telling a senior leader what they need – and a requirements document, which tells a senior leader nothing at all.

Perhaps people who love people and the real world, more than you love machines. People who love the broader picture not just the technical detail.

The software map makers will be the people who will drive change in the world and lead it to new places. All organisations should have map makers if they want software – and all software houses should have map makers.

Perhaps the formal title will be something like project manager, business analyst, or product manager.

Your role will be to generate a clear picture of what the software will do and how it will be developed, working together with stakeholders, getting everyone aligned on where you are going – and then following the map rigidly – so people can just look to the map if they want to understand how the software works.

If you think this point itself sounds abstract, consider that all cybersecurity failures can be described as a failure to build systems which support the goals of the right people while obstructing the goals of the wrong people. A failure to see the technical environment in terms of goals. A failure to abstract the technical environment into different concerns.

MORE IN-DEPTH IDEAS ABOUT SOFTWARE MAP MAKING

Introduction

As we described in the last chapter, the specific improvement we propose is far more of the work of software development is involved with in developing the map, and discussing it with other people. Less time is spent actually making code.

This map making process would involve understanding the core goals of the organisation, understanding how the key decision makers drive the company so that it achieves these goals, what mental models they use, and how that can be used as a basis for software.

The software itself should be built closely to the map, as a building company makes a building closely to an architect’s plans. Perhaps the software can be even made directly from the map (otherwise known as a ‘model’) with the emerging ‘no code’ software tools.

Maps are not technical in themselves

At this point in the book, you are probably wondering what a software map actually looks like.

You can find a Wikipedia page about software maps, which suggests making a map with boxes of different sizes, indicating the quality of code, the status of software development, or risk levels.

We propose a far less technical definition of a map. We can define it as an illustration about how we are going to achieve our goals, which is light enough on detail for people to easily understand, and yet which is heavy enough on detail to be useful.

Like the metaphorical "road map" which businesspeople and civil servants often talk about. Or, indeed, any other plan used in working life, such as a plan for a building. It contains the level of detail which is needed, but no more.

If we want a technical definition for a map we could call it "a system which emulates the real world situation in a continuous framework of relevance / relationship to support human understanding."

A challenge we see with discussions about software mapmaking is that technical people are used to talking about technical topics in public and not so comfortable talking about abstract topics like this one.

A map – as a concept – is the very opposite of technical. But, as we try to emphasise in this book, the results of software made from mapmaking approach can change the world, change profits, and lead to highly quantifiable results.

The format of the map is not important. The function is important - the map should be a means of achieving alignment between the parties and giving instruction to the programmers.

Choose the level of detail

Like with any other map, people making a software map have a choice of how much detail, or granularity, the map should contain.

You want a granularity level which enables the map to give enough information to people who have to execute on it, while not having so much information that other people involved find it hard to work with.

In the same way that a building plan gives all the information a builder needs to construct the building, and can also be used by people in the planning office to make a check about what is going to be built.

Consider how a geographical map reduces the complexity of the world we live in into a series of lines we can work with to get somewhere. But we have a choice of maps at different scales available to us. We don't need much detail to plan a long drive, we need a lot of detail to plan a building extension.

The purpose of the map is communication

The purpose of the map is to support communication and understanding between different people.

It performs the same role as language itself – a means for one person to share what is in their mind with another person.

Our minds are continually forming narratives to explain how the things that we see work. We do this subconsciously and intuitively, without willing ourselves to do it. We are highly sophisticated at this – and also don't need to think about it.

In our minds, we combine together logic and information, and can retrieve theme instantly. We remember what these facts relate to, and what makes them possible.

This is how we update each other about changes, or how we share complex information.

The process of creating these narratives could be described as 'creating abstractions' – we create a simplified picture in our minds which we can use to work with and share with others. This sharing could be made with language, maps, or some other means, and we don't really need to define any difference.

A software map could be explained in words. For example, if it is software for a metro train payment system, we could begin with the goals of the passenger, which is to have a payment system which is as fast and intuitive as possible, and the goals of the metro company, which is to reduce customers which travel without payment, and reduce the costs of managing the system. There may be fundamental choices which are made beforehand as business decisions, such as what type of ticketing or payment systems will be used.

As with real world maps and language (both written and spoken), software mapmaking can be something of an art form, with complex judgements about what to include and leave out.

Explaining abstraction levels

Perhaps the idea of 'abstraction levels' is worth more explanation. It is a concept we all use every day but do not think about it in this way. Here are some examples of how we think in different abstraction levels.

Let’s start with how we view our own lives. We see our own lives at a high abstraction level – such as our long term goals. We see our own lives at a low abstraction, or high granularity level – as a succession of tiny thoughts and actions.

We see nature at a high abstraction level when taking in a landscape, and see it at a high granularity level when understanding the different biological processes which happen.

A politician explains how the world works to people in a high abstraction level. Our political beliefs are abstracted models of how the world works – the big things we need to get right in order to have a society which functions well.

An architect operates at a high abstraction level when thinking about how a building will look like on a city’s skyline or how it will feel like to enter. An architect operates on a low abstraction / high granularity level when making plans to pass on to the civil engineers who will make the building.

We could easily describe a building design process at multiple abstraction levels, ranging from the concept of what the building will look like to placing the concrete.

A map making example – shipping maintenance software

Here is an example of how a map for software for a maritime shipping company to manage the maintenance of the ship engines could be built.

The core goals could be described as keeping ship engines operational, reducing the amount of time and cost spent on maintenance, and trying to schedule the maintenance so it can be done at convenient times.

The process is geared around the ship's main engine. So we start with that, as the first dot on the map.

We have other machines which connect to the main engine - compressors to bring in air, pumps to bring in fuel, and a propulsion system which carries the engine's power to move the ship through water. They also have to run reliably. These other pieces of equipment can be shown as dots connected to the first dot.

Each item can break over time and under stress, in which case the machine needs to be stopped for repair and waiting for spare parts. So we can show various problems connected to the dots.

Some periods of time are more convenient for doing maintenance than others. These are shown on a calendar.

The maintenance work needs people, perhaps from outside, and spare parts. These can also take time to organise. This is more dots.

There are predictions for how long parts can operate before they break, taken from experience. This can be connected with the calendar of convenient times to do maintenance, so maintenance can be planned at convenient times.

There are auditors who come onboard at certain dates to check that the maintenance has been done. So the calendar above has deadline dots.

This picture of dots is based on the mental model of someone who has actually had the task of managing ship engines, and is one of the authors of this book. And this picture of dots can be used as the basis for building software to manage all of these things.

Explaining the idea of mapping to goals

The idea of building something around goals rather than objects sounds a bit abstract in itself. This example may help explain what we mean.

On Margaret Island in Budapest, there are two places you can go to swim. One is for serious sport swimming, including racing and diving. The other is for leisure swimming, a place to take your families for lounging in the sun.

If we were planning how we were going to build the swimming pools on an objects basis, we might use the same objects for both. Both involve swimming pools, changing facilities, cash desks, showers and toilets. Perhaps there would be identical elements between the two places, and they would end up looking quite similar.

But what you find if you go to Margaret Island is that the centres are completely different – because they have been designed from top to bottom around different goals – serious sport and family relaxation.

This persists through to the details. The serious swimming facility requires swimmers to wear hairnets, because loose hairs must cause some kind of obstruction to serious swimming sport. The relaxation swimming facility has no such rules.

The open spaces around the pools are different, one for crowd spectator sport, one for sunbathing. The changing rooms are different, one more geared to young families. The catering is different.

And both swimming centres were designed in a pre-computer age. We can hypothesise how it might have been done differently – and worse – today, in our era built around object centric software. Would both centres end up with identical changing rooms, pools the same shape, regulations and features around the pools?

Software built through map-making should end up being built like the swimming pools on Margaret Island, with so many small details being aligned with the core goal.

Understanding the goals – talk to senior management

When building software to an organisation's goals, a first step is to discover what the goals are, and it may not be what you immediately think.

Is the goal of a public transport company to move vehicles according to a schedule, or move passengers? Is the goal of an oil company to sell oil, or gain access to lucrative oil reserves? Is the goal of a hotel to achieve high utilisation, or high room rates? Does a farmer want high land yields or access to high EU subsidies?

A mature organisation will already have figured out the main steps it takes to achieve its main goals, in the manner it defines them. Usually not much of this will be stated explicitly outside internal company meetings.

And all of these organisations have multiple individuals who have in-depth mental models about what needs to be done to achieve the organisation's goals and how to get there – and having discussions with others about progress towards the goals.

The software map maker will need to talk to the individuals who make these decisions which guide the company towards its goals, to ask them what they need to see to make the decisions. How their situation awareness works, what information should be presented to them, what times do they need to be alerted about a change. And this becomes the basis for the software map, which is a basis for the software.

A further benefit of these discussions is that it is a good way to keep

senior management engaged in the software – letting them talk about the world in the way they understand.

As of 2019, most of the efforts to engage senior management in software is done by trying to persuade them to watch software demonstrations, in effect trying to persuade them to accept that the software's version of their world has any relevance to the world which they hold in their heads.

Software project managers and developers commonly complain that their 'users' do not engage enough, early on, in the software development process, but instead voice their complaints after the software is built, when making changes is much more expensive. Perhaps there are some good reasons for this lack of engagement – and map making offers something of a solution.

Brick or modular elements

Another element of software maps is that it can bring in components as modules or bricks, thus simplifying the picture.

Consider a map with a symbol of a church on it. It doesn't have to give us lots of information about the church. The idea is that it is most useful for someone to know whether or not there is a church at that spot. So the church symbol is like a 'module', simplifying lots of information into the most important element.

This sort of thinking is important in software development, which usually involves lots of complex components, just like your laptop. Nobody needs to understand how all the different components work, they just need to understand how they need to fit together. This can be shown on a map.

Here's another example. Software running on cloud servers can use complex "hardware accelerators" which move the various hardware demands around to make most efficient use of the available hardware.

If you have one software application with big demands of memory, and one software application with big demands on the database, you can run them both across two servers, and each have double the database power and double the memory for each.

The hardware accelerators are themselves complex, and the way they interact with the existing software is also complex, working out what can be moved where.

But to manage this complexity, hardware accelerators are sold as a module or brick which a cloud software company can buy as an add on to their product and everything works. All the complex computation is hidden from their view.

Fitting computer power with human power

In the world of 2019, there are so many examples of people doing tasks which computers do better, such as people doing simple continuous administration tasks on a computer. There are also many examples of computers doing tasks which people do better, such as robotic call answering systems.

The way to improve the current situation would be by having systems which much more carefully integrated human strengths with machine strengths. In other words, a map.

Figuring out where computer power is most appropriate is highly complex itself. If your budget is big enough, you can program a computer to beat the world leader in the most difficult board game.

But in a task which involves quickly learning what is important in a new domain, without necessarily being taught, and without having much of a data set to learn with, a person is a thousand times better than any computer.

In between, we have domains where people do tasks repetitively, but it does not mean it is worthwhile programming a computer to do them, because not enough people are doing the same task enough times to justify the programming cost.

Computers can be programmed to follow statistical models and achieve better results than people, for example making a system to decide what medical choice leads to the best outcome based on past experience. But this only works if you have necessary budgets and expertise to build the computer system.

Machine learning software is useful for classifying or sorting a list of objects in a kind of structure, and this functionality is slowly finding more and more application across our lives, whilst simultaneously seeing many people disappointed after its capability has been over promised.

The capabilities, and relative capabilities, of people and machines are constantly changing in many ways.

But with the right sort of map, we can work out how to put it together in the best way, and keep our map continually updated.

Problems with traditional software methods

When software is made in the traditional way, with objects and relational databases, the database is the core of the software. In other words the database tells us the relationships between object and their attributes.

The code acts on the data to create what are called conversions. These could range from accepting user input, making sure user input is valid, taking user input and making changes to other data, taking newly changes data and propagating to other data fields or attributes.

Any further instructions about how the logic of the system works is provided as a text description for developers to read. Although the actual working of the system might not end up aligned with this text.

The code is written piecemeal, with small increments which are not in a continuous map with other code increments.

This means that when one code increment is complete, it changes a data field to a new value, but does not tell the developer what other code increments act on the newly changed field, so as to get a continuous view of the progress of the logic.

It’s a bit like running a factory floor where the worker has no idea who is going to ask her to do something and what that may be.

The programmers can get software working, but since code has no structure, only some labelling can help the next person understand it.

Making any subsequent changes to the database and its relationships can be very complex, because every change, such as adding a table column, has to be related to other tables. For example, if a developer adds a field such as social security number to a person's identification, the developer may need to connect it to nationality and jurisdiction, since social security numbers may vary with nationality and jurisdiction.

But then the code also needs to be adjusted to the change, which is more difficult. The code is not structured so it's not clear what increments of the code before the change rely on the old data base relationships.

For example, a piece of code to make sure a member of a large staff is correctly identified may not have included the social security number. If the new column added is intended to be part of identification, the identification logic has to be found and changed.

So adding say columns to database tables may occasionally not be a problem but in many cases the new columns may adversely affect how the logic worked before the columns were added.

So the developer has to find the code that refers to the tables where the columns wee added and make sure the logic still produces the desired outcome given the new additions. Often, it isn't easy to find this code.

So we get to a situation there the only way to know for sure what the effect of any change will be - to try it and see. The testing is not comprehensive because we cannot know every possible way the use case and logic accessed the data previously to the change and therefore how the change affects all the use cases.

The whole structure becomes very fragile, where small changes can easily destabilise the whole thing, with no-one having clear understanding why.

This results in the classic lag between a user finding a problem in how data is shown or processed, and the developers responding.

There is a classic phrase "cannot reproduce the error" often heard in the software development world. What this actually means is "we cannot understand enough about the logic to determine if such an error can be made". Perhaps because it is not easy to find and check the code which acts on the data which a user as reported a problem with.

Identification systems are hard to manage because the logic that acts on identification data could be anywhere in the code blocks. An identification system is needed to make sure the same person does not end up with several sets of identifications in the records, or does not get left out completely, and so going without a pay check.

How software map making fixes

this

If software is designed around a map, then data stores can just be another ‘node’ on the map located but the links that use the data. So you know how the data is used.

So when you add a node as a new social security number you can see how it is used for identification because it’s all localised by links in the map.

The ‘strongest’ part of the software, its core, is the way the software is designed around the organisation’s goals.

Localised goals such as making sure you identify people accurately to avoid getting people’s pay-checks mixed up, to higher level goals like getting a quality product and service by assigning the right people to the right jobs.

Even higher level goals might be knowing all your customers and their likes and dislikes, by having them serviced by the same experts as far as possible, and an even higher goal might be abstracting all the reports form the customers, to problems and opportunities in product and service improvements.

And if the company’s core goals do change, such as from a new regulatory requirement which changes how the company needs to operate in a fundamental level, the senior management can figure out how the company will work from now on, and then the software map can be updated.

Since the software is built directly from the map, it is relatively easy to see what changes are now required to be made from the software, and ensure that the resulting system will work.

You may ask why you can’t do this with the old paradigm and why don’t companies do all this today. You can theoretically, but the software architecture we just described does not help.

GETTING MORE TECHNICAL ABOUT SOFTWARE MAPS

A software map needs to be executable

For the software map-making approach to work, the software does indeed need to be built from the map. We can't have a situation where somebody makes a map and then programmers go off and do their own thing. Because then we end up with the same problems of software which nobody understands, does not necessarily deliver, and is hard to update.

Many software programmers won't like being told exactly what to do in this way. But here's an analogy which may help persuade them from the world of stadium music.

The veteran band The Who toured stadiums across the US and Canada during 2019, with a full live orchestra as an integral part of their show. But rather than have an orchestra travelling with them, they hired a local orchestra in every city they visited, and gave them sheet music.

The sheet music contains all the information that the musicians need to play the music which is required. They do not include any personal interpretation or expression, and none is expected. Any personal expression would make it impossible for the system of working with local musicians to work.

With sheet music, The Who know exactly what the orchestra is going to do at any time. The music tells the musicians exactly what they need to do at any time.

The Who's core musicians are not using sheet music and have much more freedom on stage. But they also have in-depth mental models of how the music should sound, since, in the case of the two leading musicians, have been playing the same songs for 45 years. They also created the music in the first place, using their expertise to map it against the goals of their audience at the time, 1970s teenagers, who have now grown up to be the 2019 audience. That's quite a sophisticated mental model.

The world of music in general involves a mix of freestyle playing and rigid sheet music following, with systems which have evolved over millennia and continue to involve today.

Like the world of music, software development can be a mix of map-making and map following. The question is getting the right elements in the right place.

When a map provides all the information needed to create code, in the same way sheet music provides all the information needed to make music, we can consider it 'executable'. No further figuring out is required.

We have an end result where we can imagine what the code will look like from the map, as we can imagine how music will sound like from looking at sheet music.

We can also fully understand the code, as we can fully understand music made from sheet music. (Understanding music where all the musicians are improvising is much harder.)

The vision of the mapmakers, and the people they consult with, such as the organisation's domain experts and senior leaders, is entirely realised.

For this to work, the map must include all of the detail needed to create the software.

Just as sheet music provides all the necessary information – or, as another example, a geographical map provides all the information needed for you to achieve your goal, whether driving a car across the country or planning a building extension.

The map can also ensure that the vision or request of the client – who might be someone wanting a piece of music, a building, a transportation task, a movie, or a software product – is entirely realised.

As with film making, music and construction, the more complex the development the more the personal expression and creativity should be moved upstream – to the people designing the overall project, not the people carrying it out.

Certifying security of software

The public has suffered a number of high profile computer hacks – and governments want to protect them, just as they want to make it safe to walk down the street or fly on an aeroplane.

Mapmaking can be an important component of providing this security – in the same way that mapmaking is used to certify security anywhere else, such as in an airport.

You could probably explain a map of airport security to somebody without being an airport expert. You would explain that a large area of the airport has a metaphorical ring drawn around it, called ‘airside’, and there are controls on everything which passes into the ring. The runways are included in ‘airside’.

The airport usually works on the basis that every object on any aircraft landing has passed through the same security controls in another airport, so it is safe for a person or object to be transferred from one plane to another without further scans.

We could make a plan for software security in the same way. But before we can do that, we need to actually understand what the software does.

This is very hard to do when software is made with conventional methods, which can mean that no living person actually understands one component of the software, even a component which they made themselves. So we are a long way from anybody understanding the whole thing.

A map for software cybersecurity could involve defining which assets the company has which are considered of potential interest to a hacker or at a potential threat from a hacker.

The map can include the security controls on these assets, such as passwords or checks of the source of any computer command. An objective person can make a judgement of whether these controls are appropriate to the threat, also taking into consideration cost, hassle and alternatives.

Such a map could easily handle gradations in security, for example where there is certain data which needs to be kept at a higher standard of confidentiality, such as patient records in a hospital. The organisation is willing to accept higher hassle and cost associated with this security, but without the same requirements placed on all of its data.

The map could also be easily changed if the software security requirements change, such as due to a new hacking method or a new regulation.

If the software is made rigidly to the map, as described in the previous chapter about executable software, then updating the software itself, after the map changes, should be a relatively simple task.

We are increasingly going to see regulators taking an interest in ensuring the security of software, such as in the European Union where the task is handled by a body called ENISA. If the software is made to a map, this is much easier. The regulators needs to be satisfied that the map works and the software reflects the map.

This is far easier than the regulator having to take an in-depth look at how all the software works, with no map.

Modern software cybersecurity is managed like an airport with a range of different security precautions around the “airside” line, hoping that one or other of the precautions will stop a hacker, while meanwhile giving them plenty of places to hide. Hackers thrive on the complexity and lack of understanding of modern code.

Separation of concerns

The concept which software people call “separation of concerns” is perhaps the most important in mapmaking.

All of us have an in-depth understanding of separation of concerns – because it is an important part of everyone’s life. We don’t see it as such, because we don’t need to. But in the software world, we can probably say that any time software obstructs us doing what we want, somebody has got the 'separation of concerns' wrong.

We have to talk about separation of concerns because of the why our minds jumble things up together, and the importance – at times – of un-separating them in our minds. We don’t necessarily jumble them together the way that other people do, or the wider world does, and if this is not realised, it can be a problem.

To illustrate, a single item of mail may be most important message of your life, and one of a thousand objects for a postman to deliver that day, and tasks on the postman’s mind. We can all see the difference, although this particular difference causes no problem.

Poorly understood separation of concerns is a bigger problem when people jumble all their relationships up in their minds to the extent that they cannot see that a change which seems relatively small to one person may be life changing to another.

Consider the marital affair, a situation which all of us understand. A married person believes that a switch to a new partner is a relatively small jump to make in life – one partner is exchanged for a new one, and the mental box called ‘relationship’ is never empty. The pros seem to outweigh the cons. But this change is a much bigger concern to the married partner, who goes from being in what they thought was a secure relationship to no relationship at all.

An example of how poorly separated concerns becomes a problem in the software world is when we have cybersecurity managers taking the same security recipes from one employment role to another. So the solution to every cybersecurity problem becomes the same.

There is no recognition that different companies have very different threats from hackers, different assets under target, and different costs to the company of a hack. There are companies which might have to close completely after a serious hack, and other companies which can function nearly effectively with no software at all.

We described in the opening chapter, there is a tendency of people with a software mindset, often known as ‘geeks’, to only look at the world at one abstraction level – that of high granularity. So not seeing the wood from the trees – just seeing the trees. In a fight, someone with this mindset might be immediately ready to engage in hand to hand combat, but not ready to look at whether a conflict could be avoided.

So we see security controls in the wrong places. A situation where security requirements are highly onerous and intrusive, but fail to protect us.

The way to improve cybersecurity, or indeed poorly performing software of any kind, is to take a step back, and think of what you are trying to build and run as an abstracted map, rather than look at the details. But you will only be able to do this if you are able to “separate your concerns.”

Airport designers and planners are relatively good at this. Although we complain about security hassles and delays, these are just one element of the many factors which an airport has to simultaneously manage, both as individual elements and as part of the whole, and they are becoming relatively rare, at least in the UK.

Other elements are the fast turnaround of aircraft required for the budget airline business model, the retail revenues from the airport needed for the airport’s business model, getting good utility out of the airport gates / runway / space, evening out peaks and troughs in passenger flows. The employees have their own concerns. All elements need to be manageable and simple enough.

The airline security process itself has a number of different concerns - keeping the airline free of the wrong thing, while making the security process as smooth and requiring as little thought as possible from the customer, and also having staff which do not need enormous amounts of training or skill, which would make them harder to recruit and more expensive.

The software sector does not have anywhere near as good a reputation.

A supermarket self-check-out machine needs to separately meet concerns of the supermarket (avoiding theft) and the customer (ease of use). Yet, as our experiences with them show, with alerts about the "unidentified object in the bagging area" from a poorly calibrated machine, this is not achieved.

Some of the worst examples of separation of concerns is commonly found in public transport payment systems. The system is designed to meet a basic objective of ‘providing facility to purchase a ticket’, but then the concerns of the organisation take over, such as ‘ability to identify and punish people who travel without paying’.

In the middle, typical traveller concerns such as ability to get the right ticket quickly, ability to pay with a foreign credit card, ability to make alternate arrangements when a credit card has been cancelled (as commonly happens during international travel) are often forgotten.

Another, more complex, example of poorly separated concerns in software is in software tools for making a schedule, a common software task.

In the real world, anybody making any kind of schedule will consider a wide number of elements, such as avoiding asset idle time, estimating an appropriate level of flexibility / slip room required, matching an asset with a task, what the risks are and costs of mitigating them, time taken to switch from one task to another, and how conflicts from changing situations may occur and be resolved. The software task which is easily imagined as a box of objects turns out to be very different.

The ability to separate concerns sounds like empathy, being able to see something from someone else's shoes. Certainly empathy helps, if helps you feel how someone feels, and why it is different from what was expected. But even if you are low on empathy, as many of us are, this is something you can do with logic alone.

Why we need more than UML

As of 2019, the software industry’s standard method for making maps – or models - is called “Universal Modelling Language” (UML).

We think UML has limitations if it is to be used to make models for software which meets real world goals.

The root structure of UML is built around ‘objects’ – something which can be described as a thing. The UML model shows how these objects interact. It can be called an ‘entity-relationship’ model. It is a method for modelling the interaction of things.

That has been exactly what software developers needed for most of the time until now – because the biggest and most important software projects were indeed about processes converted into things, such as bank account software.

For example, that’s how software for accounting works. Company accounting systems use objects like ‘purchase orders’, ‘invoices’ and ‘bank accounts’, and have a rigid model of how these objects interact. There are no changes expected to this model over the lifetime of the software, because company accounting system fundamentals are not expected to ever change much.

Other examples of software objects could be a ‘book’ in a system for managing books, a ‘person’ in a personnel management system, a ‘part’ in a purchasing system.

We can track this way of thinking back to the days of the first big computerisation contracts in the early 1970s. The big challenge was to make a computer system which fitted around the data records which already existed. The real world was diminished into ‘objects’ and ‘relationships’ to enable this software to be built.

But consider how much understanding got left behind in doing that.

For people who work in banking and business, an ‘invoice’ is far more than an electronic object. Issuing an invoice to a client, and having the invoice not rejected, implies an agreement between both parties that there is no further impediment to transfer of funds. It is perhaps the last step after a big mountain climb of negotiation and delivery of what was negotiated.

In the banking world, owning a bank account means having an agreement with a bank, where the bank will keep your funds safe, make them available as you need it, and perhaps pay you interest.

And in 2019, it is hard to think of an example of a company which has achieved commercial success – or other high level goals – due to how well they organised their objects. Perhaps we could have done in the 1980s, but not now.

Having a basic management software system is as fundamental for business today, as it is for a restaurant to have the fundamentals – a kitchen, tables, staff, menus, food in the fridge. Just possessing these elements is not what makes a restaurant successful.

So this leads to our suggestion that our basis for building software is a map of the goals of the organisation, rather than objects.

A justification for object centric software development methods we sometimes hear is that the real world is in fact constructed around objects, or things which can be described as objects, such as ‘investment’, ‘sale’ or ‘building’.

It is true that it is possible to look at much of the real world in terms of objects.

But this is rarely how the people working with these things understand them. If you work with these objects in your daily life, you will see them as far more than objects. An investment is something in which an investor places a lot of careful consideration. A sale is a culmination of much effort by a salesperson. A building, to a landlord, is a source of income which must be balanced against its costs.

A great deal of effort is happening in 2019 to build software to support these things using object modelling. We hear about projects like ‘swim lanes’ to build processes around objects.

But look at the trends – the demands from software are getting higher, and further and further away from a world which can be simply described with objects. Maybe the object centric approach has reached its limit.

In the real world of 2019, companies who want to use software to achieve competitive advantage want to use software in ways which are hard to describe with objects.

For example, an electrical utility which wants to align their communication methods across multiple business units. A shipping company which wants software which is far more aligned with the real company goals. A metro operating company which wants customer payment systems which are very easy and fast to use.

Beyond the software world, we have lawyers who want to win cases, business people who want to grow their businesses, school teachers who want their classes and schools to thrive, regulators who want to reduce certain types of crime, marketers who want successful marketing, police who want to keep their streets safe, hospital managers want to maintain a good service to their communities.

Even when we do a project completely in the realm of digital technology – such as the electrical utility aligning the communications between different business units described above – we can do it better by understanding the different goals of the individuals and companies involved and finding a way to map them together, rather than modelling it as objects.

You shouldn't need to work out the logic yourself (as a programmer)

The logic for software can get really complicated and it usually falls to the programmer to figure it out. It shouldn't be this way - the logic should be figured out in advance, as part of the map. But that's not usually how it happens today.

To illustrate, let's consider we are making a software system to schedule people to tasks. It could be a headteacher matching the right teacher to the right class, or a plumbing service company allocating plumbers to jobs that day, or many other examples which all have similar challenges.

Perhaps there are some tasks which only one person can do, such as a teacher with skills needed for a special class, or a plumber with specific skills for a certain job. Perhaps there are more highly skilled people who are suitable for particularly challenging tasks, such as an advanced physics lesson or a major plumbing installation. There will probably be a range of different tasks and a range of different skillsets in your expert 'pool'.

You may also want to have some learning in your system, identifying where you have a deficit of skills, and giving people an opportunity to learn those skills from someone who already has them. People may have personal preferences about what tasks they want to do. There may be some definitions of a 'standard' expert, as there is for pilots and doctors to some degree, which would make the scheduling simpler, but not simple.

All of this would be taken into consideration by a human scheduler, such as a school headteacher, and other people would take it for granted that this is done. But if the scheduling is being done in software, then logic has to be written to take all the above elements into account and more.

A similar challenge from a technical perspective would be a system to gather together customer feedback on a software product, working out which requests are being sent by multiple customers, which requests are being sent by the high value customers, which requests are relatively easy to build, which requests you as a software company agree will improve the product enough to justify the effort, and so select the updates you will make and then schedule a programmer to build them.

So how would this scheduling software be built? In conventional software development, the work would start by defining databases, data entities, and making a model of the objects. So, you might have objects for tasks, experts, skills.

But it would be very difficult to include all the details of the logic in UML. This is because the real world task is not about objects interacting, it is about finding the best way to achieve a goal. And UML is all about how objects interact. UML can include flow charts and diagrams to try to indicate how real world logic works, but not as a pathway to writing code. Defining real world goals based on objects and interaction between objects gets very complex very quickly.

Doing a scheduling task can end up with very complex processing at the level of taking data out of databases. A simple task like working out how much salary to pay staff involves one table of members of staff names and another corresponding one with their salary scale, another table of how salary scales correspond to salary and more. This is just to retrieve the right data to start applying logic to make appropriate conversions.

So in conventional software development, we end up with software programmers themselves working out what data to retrieve in order to apply logic to it. Then working out what logical sequences will satisfy the requirements. This is after being handed a bundle of requirements documents, entity-relationship models and perhaps some UML diagrams.

The database and logic are built as two separate worlds. They do not sufficiently correspond to make the logic a model like a map. The programmer's focus is on making something which actually works, rather than being concerned with how lucidly the result is actually achieved, such as keeping a sequence of steps showing exactly the salary is calculated by the computer. The Entity Relationship Model, if there is one, will define how different objects relate, but not the process of working with the attributes contained in the objects to achieve a goal.

So it is not surprising that the code and logic ends up being unstructured, and more like switching between 10 different pages of sheet music at the same time, causing problems when it is being checked, tested and improved.

In our vision of how this would work, all of the logic would be included as part of the map, so there is nothing for the programmer to have to work out at all. The programmer just has to translate the written logic into something the computer understands. There is no need to provide code as to of what data is drawn from the database as this data corresponds to the nodes directly adjacent to the logic. The programmer only needs to write in a computer readable language how the data in the corresponding node is "converted" to get the desired result. Both the database and the logic are constructed according to the map, so they both work together and can show the logical steps in sequence.

There is no need to make an entity relationship database, no confusion about what has been coded and what hasn't, no need to make other conceptual models to try to see how the real world needs will fit around the software functionality. The software programmer's role is just building what is shown in the map.

This brings us back again to the discussion about sheet music. Sheet music is a notation system designed to gives a musician all the information they need to perform the music, no matter how complex the music is.

The musician does not need to think about how the music should work, and everybody is able to understand how the music will work, without needing to hear it played (or 'executed', in software language). And everybody is happy with the situation.

Could we design and communicate software logic in the same way, with a notation describing exactly how the software will work, built as part of the map, so the programmer does not need to work it out at all?

We don't necessarily need to invent a complex notation – we can describe logic using normal language, or with dots and lines on paper (as in the ship maintenance example earlier in this book). The important point is that the logic is described precisely enough in whatever notation system is used that no further interpretation is required.

Case study from banking – where objects are restrictive

In the banking world, a lot of careful thinking has been done into how the bank account works, including what is acceptable as a source of incoming funds, how information is provided in response to a 'balance enquiry', how funds are reserved for different payments although they are not yet paid, and how the timing of different transactions works, to avoid conflicts such as the same funds being allocated for two different payments.

The computer understands none of this – it only behaves as it was programmed to behave, following logic figured out decades ago in many cases.

For example, the computer knows the bank account number and your name, which it ties together with details on your bank card, enabling it to authorise release of funds when you want to make a payment, and update its records of your funds available.

The computer system for banks could be described as a simplification of way a bank works, reduced so that software engineers could build computer systems to automate it.

When the software systems were first built – in around the 1970s - the senior bankers – who understood banking – did not see a need to pass on all this knowledge to software engineers, just give them knowledge they needed to build this with.

That leads to the situation today where we have banking software built in rigid ways to handle the fundamental elements of handling funds and records, which works. We can say the core competence of a bank today is its huge scale transaction management capability.

But banks today also have systems which no-one dare change, because no-one understands any more how the software maps to real world processes. No-one knows exactly how it works. It is not just the banking software architecture which was written in the 1970s. In many cases, banks are still using actual software developed in the 1970s.

Meanwhile, many people are developing entirely new business models in the financial world, supporting much more flexible and efficient investing and lending. This needs the transaction capability of banks, but are also businesses which banks struggle to provide.

Another change happening today is that banks are no longer the only way to keep funds and fund records safe. This was one of the original reasons for having a bank. We are seeing big developments with technologies like blockchain to do the same thing.

So the rigid structure of banking software means that banks are struggling to be the leaders in financial technology today. Perhaps it is time for a complete rethink about how their software is built.

Case study from procurement software

Here's an illustration of how the inflexibility of object centric software can be a problem.

Consider a company which wants to track its spending separately for purchasing supplies and transportation of its supplies.

The company wants to better understand if it is overspending on transportation of supplies, and so if it should be allocating more effort into reducing transport costs.

But the supplier might send an invoice including transportation costs for multiple items bundled together. And the transport cost may vary with the weight or volume of the item.

Conventional procurement software is designed to support the entering of invoices exactly as they are received from a supplier, not splitting them up. So it makes it very hard to allocate the costs of supplies and transport separately, in conventional procurement or accounting software.

So it is hard for management to get a true picture of what they are spending, and how it might be improved by reorganising transport cost allocation.

And if the company wants to make split the invoices after the invoices have been entered, this can be very awkward to do with conventional accounts software, which has been designed to make it impossible to cancel entered invoices for fraud reasons.

These problems could be resolved by redesigning the objects and the associated logic.

But redesigning the objects and their relationships is not easy, revising the logic presents another problem. The problem is made harder by the fact that the code is unstructured. It is hard to modify the logic without running into unexpected knock on effects.

Case study from shipping software

Earlier in this book, we presented an example of a software map, showing how a map would be used to make software for managing maintenance of ship engines.

Now we can look at how we might do it if we were forced to use UML and define it around objects.

We might create objects for maintenance tasks, parts, reliability, expected time between failure, machines, supporting machines, resources, schedule intervals, ships, and someone would make a model about how all of these connect together.

This model of objects and connections would need to work with every real world scenario to be useful.

And it would get quickly snared by a common real world scenario, such as identical spare parts with different part numbers and different prices. The more expensive part, of course, is the one provided by the original manufacturer, with a part number already in your database. This manufacturer knows you may prefer to pay more for a part you already know will work, rather than take on the effort and risk of a different part.

Your computerised purchasing system would just generate an order for the expensive part. This is because the part number is the “root” identification of the part “object”, and there is only one available manufacturer.

The object based software could not be easily changed so that it is now defining spare parts by the goal that spare part has, and the required dimensions, strength, or other specification it needs to achieve it, rather than by a part number.

Objects just have bureaucratic status attributes, not goal failure relationships. The ship 'object' does not include any understanding of what the ship is doing, and the 'machine' object does not include any understanding of the role of propulsion in the shipping industry.

This model of objects and connections could not be geared around any real world goal (such as avoiding a failure of such significance that it takes the vessel out of service).

It might be geared around a perfect world as defined by the object software, such as having all the maintenance tasks done according to schedule.

DEVELOPING NEW SOFTWARE COMMERCIAL MODELS

The big challenge with the software mapmaking approach is that the dominant commercial models for software are not aligned with it. This does not make the map making approach impossible, but it certainly makes it more difficult.

But also the current commercial models are arguably not well suited to what society needs from software today. This will force them to change. But they may change more slowly than we would like.

Let's say there two current dominant commercial models – the big contractor model and the venture capital model. We will look at each of them in depth.

The big contractor model

The big contractor model involves a big software contractor making software for one large client.

The work process generally starts with a client compiling detailed documents of exactly what the contractor will build, called "requirements documents". Different contractors will bid on who can build to the requirements.

The client may decide that the prices are too high, and much simplify the requirements. Eventually there is an agreement between client and contractor, with requirements which fits in the budget, precisely defined in the documents, and work begins.

These "requirements" documents are not maps. They are more usually a list of demanded features, written down by people who don't have much understanding of software. People who are unaware of the goal/process elements missing from the requirements, and are often unaware how different the result could be from what they had imagined.

The requirements documents are not necessarily made with enough understanding of the company's hundreds of overlapping goals and obstacles. It commonly ends up with detailed descriptions of buttons and interfaces as presented by the vendor, but with little understanding of how processes and even objects interact.

So the documentation is vast and still falls short. But it's enough for the contractor to build precisely what they have been asked to build, and then send their bill. If the client isn't happy with it, the client has no substantial recourse. But that doesn't mean it works for the client.

The requirements document is also not written in a way which a domain expert would understand. By a domain expert we mean the person who actually understands how the company works, and how it satisfies its complex goals and avoids conflicts, such as a senior manager.

If a company's goals and obstacles were simple, then the process of making the requirements would indeed be simple. Perhaps a traditional software problem like managing straight forward transactions. But not a problem which involves helping people meet goals or make decisions in complex situations. Nor meeting difficult trade-offs between ease of use and too much flexibility. In short, the 'requirements' system is designed for 1970s software developments, not the needs of today.

The "requirements" model can be workable where there are big budgets involved in the software, and a high tolerance of failure to achieve desired results.

The venture capital and start-ups model

The "venture capital and start-ups" model is usually designed around the hope that a software company can sell a product or service to millions of users in a business that no one has done before. So the investor sees that the prospect has potential for a reward much greater than the development costs.

The investor knows the odds of a success may be lower than 1 in 20. But the returns from the one winner are still high enough for it to make sense.

This game has been refined over the years from an investor perspective. They know they want a product which does one narrowly defined thing better than before, for potentially billions of customers, as we saw with WhatsApp and Instagram.

Investors are also looking for faster ways to filter companies out, demanding that software companies throw together a prototype working model as fast as they can so it can be tested for its potential as a 'killer application'. In this rush of 'hackathons' no records are kept of how the software is made.

This approach passes over many markets which are considered not large enough or which cannot be satisfied with a product with a single function, or which might want a single software product to provide value for decades. So it passes over just about all of working life.

For example, a software customer in education or police would need a software tool which could deliver a wide range of functionality although carefully modelled over the specific circumstances, the needs of local children or the local police district.

Other commercial problems

Other commercial problems with mapmaking software are that software is often sold in 2019 by impressing clients with artificial intelligence capability or the promise that the software will do a lot of their company's work. These promises may not be realistic. But map making software is not able to make such promises.

The skills needed – mainly abstracting what is going to be built and discussing the abstractions with stakeholders before construction starts – are not in big supply. And as we discussed, software people usually seem to prefer to spend their time on the details of making code work, not making and sharing abstractions.

HOW TO MAKE IT WORK COMMERCIALLY

But we’ve reached a point in technology development where this sort of software development must come next. The existing software development methods and business models are reaching their limits. Society is nowhere near the limit of how much software can potentially do to help.

There was no obvious business model for motorways, the internet itself, large buildings, book publishing, or even software development when it started – yet people found a way of doing it.

The biggest beneficiaries of the sort of software his book describes may well be the weaker people in society - disabled, mentally ill, people who suffer from abuse.

If these people have no family or charities to care for them, they are dependent on governments, with limited resources and millions of people to serve.

If software can help co-ordinate these resources – people and things – to achieve 20 per cent more output from them, it would mean a huge benefit to the whole of society. Better software could achieve further benefits by helping more serious problems to be identified much faster.

Further software benefits could be achieved from giving the weaker members of our society much better quality of life, through better managing the maintenance of their housing, for example, leading to fewer health problems.

Better software would provide more benefits for the people who work in the sector, helping them have much more pleasant work, such from enabling them to support the same individuals every day, rather than having a life of 5 minute interactions with strangers.

We can find better ways to keep track of the various disabilities or other problems people have, so they only need to be assessed once and by someone capable of doing it

We can find better ways to keep track of what is working in terms of improving people's situations.

We can find better ways to organise voluntary workers so that they can fill the gaps without leaving any holes.

Better software can also more quickly identify problems which can create huge costs in our social systems, such as from fraud and crime.

In the environmental / energy sector, better software could make a big contribution to reducing CO2 emissions and climate change. CO2 emissions are a result of fuel combustion, and reduced by reducing this fuel combustion. We do this by using fuel as efficiently as possible.

The most ineffective use of fuel is when it is used to give us flexibility, such as to power a car to carry us somewhere we could go by train or bus. The same with goods transportation.

As of 2019, road vehicles and small vans offer far more flexibility than rail for moving people and goods in most situations. But this doesn't necessarily have to be the case. Flexibility does not necessarily mean having a vehicle at our personal command, it can also mean having the option to go where we want to go and when, something which rail transport could provide much better than it does today, with better software.

Another major avoidable source of CO2 is heating. Our society separately generates heat and wastes heat. Better software could keep the heat sources and sinks must better aligned.

In the immediate future, the increasing drive from regulators to certify cybersecurity drives a need for software made from maps. Regulators and certifying bodies will need to be able to clearly see how the software works. There will be rewards to software companies which can make it easier for them. Mapping is surely the easiest way to do this. The map is a plan for how the software works, and the software must be built according to the map.

And the software industry is not dependent on big projects / contractor models or venture capital. Just like any other business sector, it can thrive by having small numbers of clients well served, providing steady business in return for delivering products which meet a client's need.

A new psychology

That brings us to the core theme of this book - that the skills needed to adapt software to the needs of society are inverse to the skills needed to make code.

And the skills to make code are not the ones which will create the new software. We need programmers to learn the new skills, and people who are not programmers to learn the new skills and also learn about software.

Making code demands nitty gritty thinking about the details, seeing the world in the same way as the computer does and shutting the human world out.

Adapting software to the needs of society demands seeing the broader picture, abstracting, seeing elements at multiple levels, consulting and discussing.

We may offend some people by saying that the era of the geek is over. We may also attract some attention by saying that. These ideas need attention. Hopefully this risk is worth taking.

The tech world's leaders, such as Jeff Bezos, Steve Jobs and Bill Gates, had software mapmaking skills in spades. They could see where their companies needed to go and how to get there. Although these skills are not much talked about because their importance has not been much recognised until now.

This is the door which is opening up for you if you want to walk through it.

Software map making club

We (the authors) are setting up a software mapmaking club to discuss better ways to work with maps in software – including with events in Athens and perhaps in London. You can sign up at www.softwarefordomainexperts.com. We look forward to seeing you soon.

ABOUT THE AUTHORS

Karl Jeffery is editor / conference producer with Digital Energy Journal, focussing on digital technology in upstream oil and gas, and Digital Ship, focussing on digital technology in shipping. He is also publisher of Tanker Operator magazine, Carbon Capture Journal, and producer of the Finding Petroleum series of events.

Dimitris Lyras is a member of a family with more than 150 years of experience in shipping, He is director of Lyras Shipping, and director and founder of Ulysses Systems, a software company specializing in the delivery of actionable information, and Ulysses Learning, a simulation-based e-Learning company.