Techniques

Legacy Modernization: Techniques

How to deal with an old codebase

In this lesson we're going to see the different techniques that we can use to perform legacy modernization.

Techniques

We are going to see the different techniques
to deal with legacy code

So we are going to see the different techniques that we can use to deal with legacy code and solve the issues that we are facing because our code is legacy and also capture the opportunities for improvements that we have.

Techniques

In this lesson on techniques, we will briefly explain what they are about. Later we are going to teach you how to apply each one of them. So at the end of any technique that will be explained in this lesson, you will learn what you can get out of it. While you will learn later how to actually apply it. I will ask you to pay attention to each technique. Stop the video for a few moments after each one of them and think about what it can do for you.

For example, one time migration will allow you to get read of an old language. Try asking yourself, do I think this one will help me? If the answer is yes, why? If the answer is not, why not?

This is the first step of your journey in legacy modernization because we can give you suggestions, we can share what we have seen in general happening with our clients and in the field. But in the end it's up to you to figure out how to apply this to your situation, and all we can do is share knowledge and give you suggestion for your reflections.

Many of our consulting clients reach us at this point. They see what is possible and how they think that they know what they need to solve their legacy problems. So they contact us. We

found out that when we explain the stronger points of each technique, some of them change their minds, some clients change their mind and pick cannot approach.

So we are trying to package that experience in this course. This course will be in a way, replace some of these discussions, so that you can understand on your own what legacy modernization can do for you and which particular solution will make the most sense for you.

And you have the advantage of knowing extremely well your situation more than we will ever. So if we are able to transmit this knowledge and you are the one figuring out which techniques make sense for you, I think you know you are in a strong position to make a very good choice. In our experience, people learn more when they have always in mind their specific ideas, and needs, and so they are not learning about techniques and solution in the abstract, but keeping in the back of their mind out that is meaningful for them. So this is why I am stressing the importance of thinking about your specific case.

So when we present something rather than just thinking, oh, this is interesting, this is neat, you can compare this and see how it fits with your specific situation, with the problems you're experiencing, and so it will help you understand better how this can actually help you, doing a little bit of extra work and be a little more focused on your specific case.

Techniques

How can you solve the riddle of legacy code?

How can you keep the value, but reduce the cost?

There are different strategies.

The goal, as we said before, is to keep the value that is throughout in legacy code, but reducing the associated cost there due to the form in which this value is structured.

Techniques

We want to approach the problem by putting together business, technical, and organizational considerations; all of which are crucial to an effective solution.

This makes the matter quite complex, it will take the rest of the course to drill down through the subject.

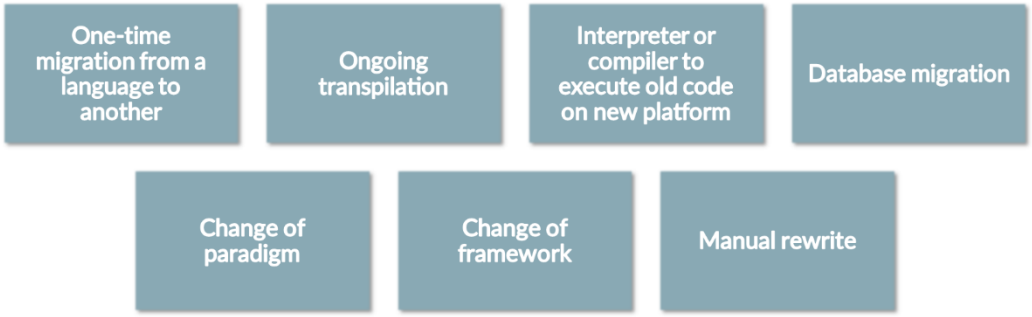
Now to identify the right legacy modernization technique, we need to consider the problem from different point of view, from the point of view of the business, from a technical point of view, and also make organizational considerations, because all of these aspects influence our choice. What makes really complex to perform this kind of modification is exactly the fact that these three different aspects are mixed. They cannot be considered on their own.

Techniques

We are going to start by providing a list of the techniques, which is the focus of this lesson.

First of all, we are going to see a list of techniques that are available and this is the goal of this lesson.

Techniques



These are the seven legacy modernization techniques that we are going to study as part of this course. The first one is a one-time migration from a language to another.
We have a legacy code base express in one language, we translate it into another language, and now we keep evolving the code base in this new other language.
The second technique, is the ongoing translation. In this case, our developers write code in a certain language. We translate it behind the scene in a separate language that we compile or interpret, but we keep writing code in the original legacy language.
The third technique, is to acquire or create an interpreter, or a compiler that permit to execute all code on a new platform for which before we didn't have an interpreter or a compiler.
The fourth technique, is a database migration.
The fifth technique, is a change of paradigm. For example, moving from desktop applications to web application.
The sixth technique, is a change of framework. For example, moving from Django to Flask.
And the seven technique, is a manual rewrite.
So I open my old code base, I open on the side an IDE for a new programming language. I look at what I have and I try to replicate it in the new language.

Techniques

These techniques represent solutions that balance the different needs your company might have.

These techniques represent solutions that balance the different needs your company may have.

Techniques

Later in the course, we are going to see the techniques in detail.

Right now we present a brief overview to explain what they are.

Later in the course. We are going to see each of these techniques in details and discuss when they should be applied first, and then how they can be applied.

One-time Migration

A one-time migration from one language to another consists in transforming all the code you have in one language into another.

It is the right solution when you want to get rid of the old language and platform.

Now we want just to give you a quick overview of the different techniques. I already spent a few words, but let's revisit them. The first one, one time migration. In this case, the idea is that we take all the code that you have written in a certain language, let's say COBOL. We translate it all to a new language, let's say Java. Then we throw away the COBOL code base, and from now on we just program in Java, like we never develop any code in COBOL before.
So it also means that we need to employ of course Java developers. And at this point for that particular module, we don't need anymore COBOL developers.

One-time Migration

With this solution, you end up using only the new language and removing the old one from your codebase.

There is another technique in which you keep using both languages.

So it's a one time migration because once we have done the migration, we have transition to this brave new world and we can forget whatever were the skills that we needed before whatever were the constraints of the previous platform. Now we can continue developing our application in this case, in Java.

Ongoing Transpilation

A transpiler allows you to keep writing code in one language but translate it at the time of execution.

Ongoing transpilation is necessary when you want to keep writing code in one language but execute it in another one.

Now, an ongoing translation is similar from a certain point of view because it also involve a transpilers, but it's different in the sense that did is that we will write code in this old language, for example, Kabul and just translated behind the scene to a new language, for example, Java and execute the Java code. Now maybe an example where we have used these techniques is with VBA and C++.

So we have a client that wanted to keep developing coding in VBA because the code in VBA was written not by proper developers, but from financial experts will have learned how to use VBA. It was a reasonably simple language. The problem was with performance. So in that case, we wrote a transpiler that behind the scene translated VBA to C++ compile it, and executed which much better performance than the original VBA code. So in this case we talk about an ongoing transpilation because this transpilation kept being performed over and over every time someone changed the code in the old language, code in a new language, re-expressing in the same element and we keep using our transpiler. So in that case, we use it and maintain it for a certain time.

Ongoing Transpilation

This is different from one-time migration because you are using both languages, at different times.

The difference is that we keep writing code in your language, in this case VBA.

Ongoing Transpilation

For example, you want to keep writing code in VBA but want to execute it on the web, so you need to transpile it to JavaScript.

And yeah, this is the sample I was referring to.

This is a slightly different situation because this was a situation in which the reason for transpiling from VBA was not for performance, like in the case that we transpired to C++ but was for executing the code in the browser. So in that case we transpired to JavaScript.

Interpreter or Compiler

An interpreter or compiler makes it possible to execute the old code (almost) unchanged in a new environment.

An interpreter or compiler makes sense when you want to keep the old code exactly as it is, but running on a new platform.

The third technique is about obtaining or developing an interpreter or a compiler that let me execute my legacy code on a new platform that was previously not support. For example, for a client of ours, we have developed, several years ago, an interpreter for RPG that permitted to execute this code on the Java tool machine. While originally they could run the RPG code only on proprietary IBM machines. With this interpreter, they were able to run RPG code inside the Java virtual machine with the possibility of integrating with other applications running on the Java virtual machine. So application written in Java code and Scala, Groovy. And so in this case you keep writing code in the same language as before, but you can execute it in different contexts, and in some cases also get interoperability with different elements like in this case or we develop an interpreter running on the Java virtual machine, we were able to pull from RPG Java classes and from Java classes RPG code and so this led to a certain number of benefits.

Interpreter or Compiler

For example, you want to stop using RPG on the mainframe and keep old RPG code, but run it on a standard server. You can then deploy it on the cloud.

Another possible advantage is that in this case, for example, you could also move to the cloud where there is support for the JVM.

Database Migration

Changing database consists in moving your data to a different database.

A database migration works when you cannot satisfy operations (e.g., lower costs) or technical needs (e.g., analytical power) on your current database.

The fourth technique is a database migration technique. The idea is that you want to move from a certain database to another. Typically, most common cases that the database that you're using is particularly expensive and so you would like to move to a different one that is cheaper, or you may have performance that are not good enough or maybe it will be not that well-supported. This typically required to adapt your code as, very frequently, different databases supports different dialects of SQL or they may use other languages different from SQL. So in that case it's even harder and so you typically need to adapt your code. You cannot just take it and run it on a new database.

Database Migration

For example, you want to move from an Oracle database to Snowflake.

One typical situation that we encountered is people that are looking to move away from Oracle databases to Snowflake or for example, also from Teradata to Snowflake.

Change of Paradigm

Change of paradigm means changing the kind of application you write, e.g., from desktop to mobile. The whole architecture might change.

A change of paradigm is the best solution when you need to adapt to a new environment.

Another technique is the change of paradigm. We have seen several in the last 15 years as people first wanted to move from console application to desktop application, then from desktop application to web application or maybe to mobile applications.

Change of Paradigm

> In the past, it might have been transforming console apps into desktop apps. Nowadays, it consists mostly of changing desktop apps to the web or mobile apps.

This typically requires a change in the architecture. It's not trivial because also the way in which the different components within the application collaborate changes. Also, the UI that we can provide can change.
Just to give you a simple example, if you have a desktop application, maybe you have a certain layout, horizontal, you have a certain size, the screen, so you can put a certain amount of component. If you translate that screen, to a screen running on the mobile, typically the layout is changed. So most frequently vertical and the number of elements that can fit on a single screen are less. So it requires a certain amount of changes.

Change of Framework

A change of framework involves a substantial, incompatible upgrade from one founding framework to another.

A change of framework is the right solution when the overall software works fine, but you need to perform an upgrade that requires substantial changes.

Change of framework is instead the case in which you stick to the same language. You just want to move from a certain framework to another, for example, from ANTLRv2 to ANTLRv4, from Django to Flask and the idea is that yes, you want to keep the same language just changing framework, maybe changing across different versions of the same framework when they are significantly different or replacing one particular framework with a framework that is performing more or less the same task, but maybe this framework is better maintained, it's more widely used, it has features that you need. And so that could be a set of reasons for wanting to swap the framework that you initially use with a new framework.

Change of Framework

The term framework is quite inclusive.

So this technique refers to a large framework, like upgrading an application from .NET Framework to .NET 6, or web frameworks, like from AngularJS to Angular.

In this case, the term framework could indicate different things. For example, updating from the .NET framework to .NET 6 or from web frameworks like AngularJS to Angular. So where this version are significantly different.

Manual Rewrite

A manual rewrite is the right answer to a radical mismatch between your code and current business or technical needs.

Automatic migration might be difficult, the codebase is small or not that valuable to begin with.

Any other approach would have uncertain results and costs.

Another techniques is a manual rewrite, so the idea is that you have a code base written in a certain language. You don't try to save it, you just rewrite it from scratch. Now this is the techniques that maybe most people are familiar with, is not the only techniques. There are cases in which make sense to use it, but we believe is overused and there are significant difficulties with this.
There is high uncertainty about the results and the cost. We have seen several cases in which someone started a manual rewrite and then they have to abandon it because it was more expensive that they expected. However, there are also advantages in this case, because it permits to working the application and so this is one valid technique in certain contexts, and so we are going to present it as part of this course.

Manual Rewrite

For example, your business has shifted from selling a plugin for doing data analysis on Excel sheets to selling a web service for data analysis.

You would like your users to keep using the same formulas they are accustomed to, but the overall architecture is so different that you need to start from scratch.

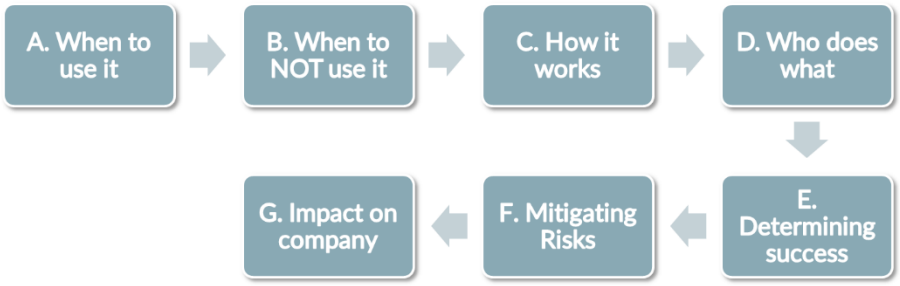
In general, manual rewrite makes the most sense when the change is very significant and the code base is relatively small, so automatize the translation process would make less sense.

Roadmap

Each technique will have its own lesson that will follow this base roadmap.

Now we have seen brief introduction of the seven techniques that we are going to study as part of this course. Now we're going to see the roadmap that we will follow for each single techniques.

Roadmap



Roadmap for a technique lesson

For each technique. We will study when to use it, when not to use it, how it work, who does what, determining success, so ensuring that the techniques help us reaching our goals, mitigating the risk involved, and study the impact on the company or the organization.

Summary

We have seen all the available solutions for legacy modernization.

Next, we are going to show how to pick among these the right one for you.

Now we have seen all the different available solution for legacy modernization. The next step will be for us to understand when we need to use each one of these.