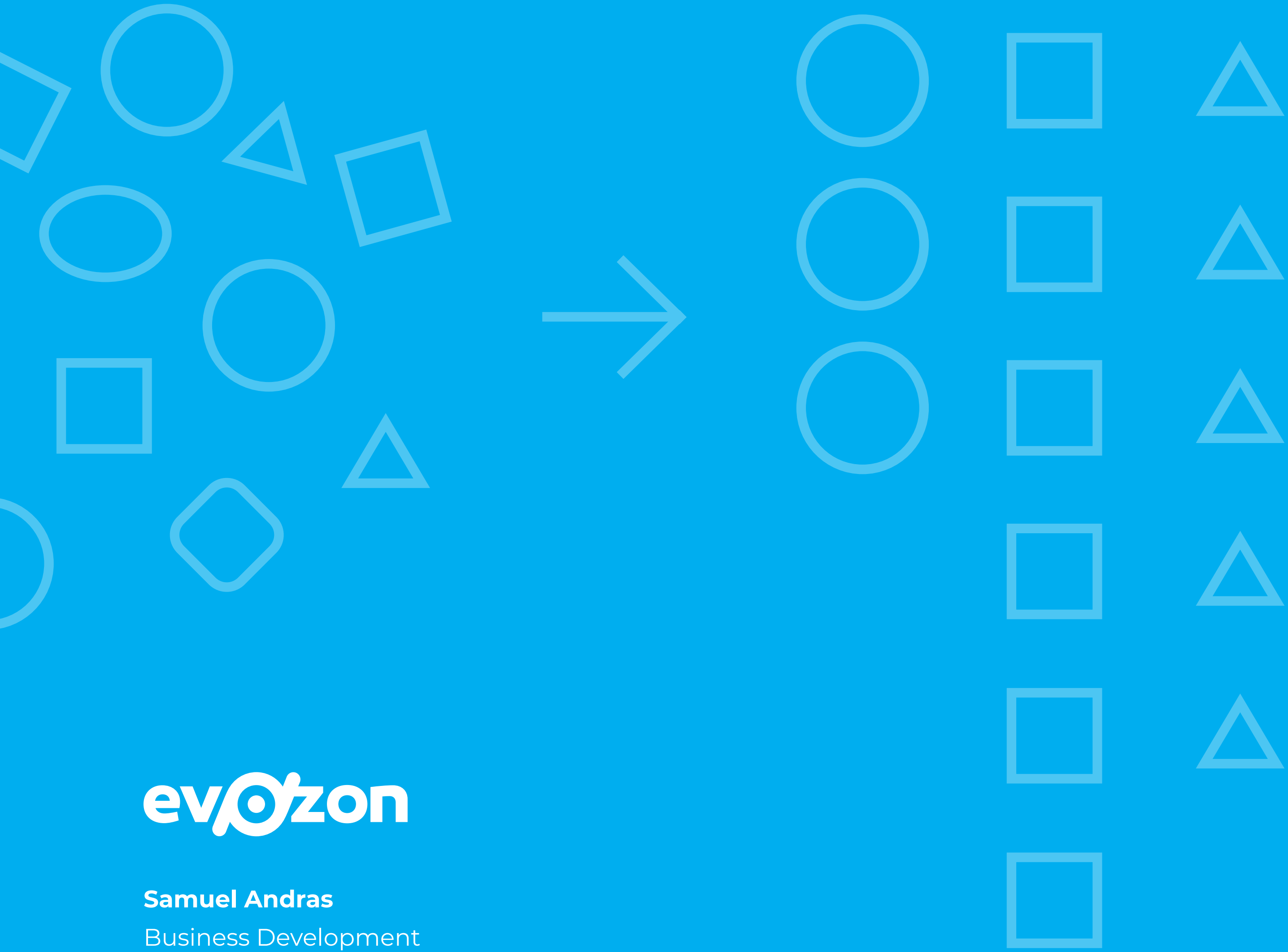
Tackling legacy projects; an experience in Perl



Samuel Andras
Business Development

Intro

Legacy code is a frightful term, one that most developers resent, and for good reason. However, there are several misconceptions about legacy code in general, misconceptions that we'd like to clear and offer a different perspective on a topic that is a big part of the tech scene. Let's be frank, everybody works on a legacy project at one point in time, so it might help looking at it a bit differently.

Going from causes to solutions, through the tunnel of legacy code, we also explore how Perl fits in the legacy scene, covering a case study for a legacy project that required some grunt work.

Whether you are involved with a legacy project, or if you will be involved in the future, this ebook will help flesh out a different outlook on the concept itself, Perl projects included.

This ebook is for developers, testers, managers and business individuals working on tech projects that are or will become legacy.

Many thanks to all my colleagues at Evozon who contributed to this ebook.

Contents

Intro 2

Legacy Code 4

Causes 6

Perspective 10

Consequences and solutions 11

Rewriting and refactoring 12

Perl 13

Case study: Perl legacy project rewrite 14

Future legacy 16

Legacy Code

Because of the different perceptions of the notion itself, there are lots of definitions for legacy code.

Here are a few definitions we put together to encompass the main characteristics and complexity of the term legacy:

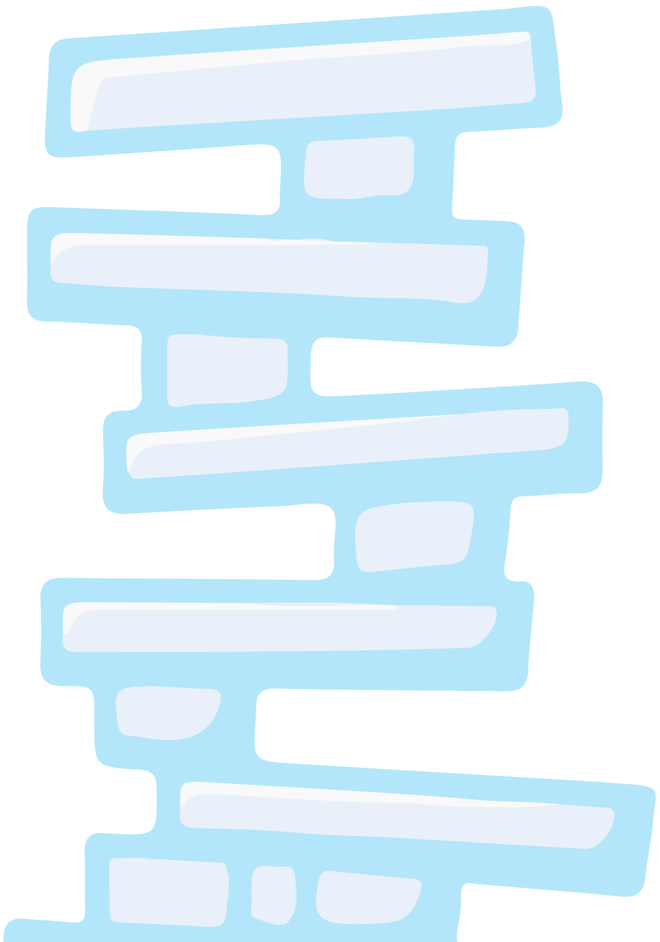
> Code which is unsupported, undocumented, that does not respect (current) programming best practices. A codebase that is or was constantly patched, rather than coded based on a carefully laid out plan with clear requirements.

> Code that was written by someone else with another mindset, different standards, using a different methodology or architecture. However, legacy code can also be your own code, written under different circumstances.

> Code which was not tested, that has unknown dependencies tied to it.

Although the word legacy has a positive meaning, if you add code at the end of it, you get something filled with negativity. One thing that should be mentioned, among all the negativity, is that legacy code is production code that works. It may be ugly, scrambled and hard to comprehend, but it works.

If you have to work with it, make changes, add new features, that's another story. It's a situation similar to a Jenga tower - you have to be careful what you do, because if you make a wrong move - it all comes towering down on you.



However, most tech projects are under one form or another of legacy, it really depends on the state of the project. That being said, it's a bit surprising that although there are quite a few developer surveys out there, the topic of legacy, or rather the importance of having a legacy or a greenfield project in a new job does not come up.

Compensation, languages, frameworks, technologies and opportunities for development take the top 3 spots in the 2018 Stack Overflow survey, which, further analyzed might correlate with what legacy and greenfield projects have to offer, but the actual question of greenfield or legacy is rarely asked in the survey format. There is no doubt that the state of the code is a factor in taking on a project, but considering the ratio of greenfield to legacy projects, the question might be more in the lines of "How legacy is it?".

Legacy code is a very comprehensive term, that includes different types of code, so offering a general definition is a challenge.

HOW LEGACY **IS IT**

Causes

What turns code into legacy?
Here are several causes that turn code into legacy code. Depending on your definition you might add a few things to this list. These apply to all programming languages, including Perl.

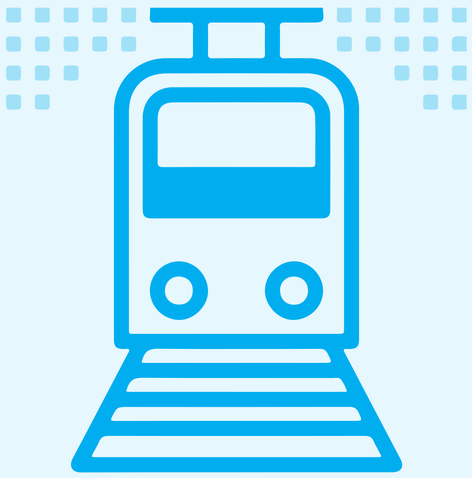
- Different programming standards, using monoliths
- Hacks and shortcuts due to time pressure
- Developer turnover and loss of knowledge
- Lack of documentation
- Lack of code reviews
- Lack of senior developers
- Unsupported versions
- Unsupported third party modules
- More features that increase complexity beyond what the original system was designed for
- Shoddy programming
- Bad programming practices

There is a perception that all legacy code is old and that old code means bad or ugly code. An important thing to mention here is that code doesn't age, it doesn't show signs of decrepitude, it doesn't gather dust, rust or go bad simply by the passing of time. If you leave it alone for a decade it will still work the same way.

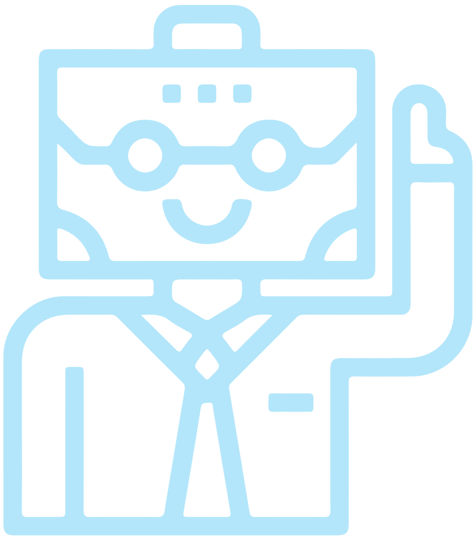
However, changes happen - technology moves forward, new versions of software are no longer compatible with the original code, new tools come into the scene, businesses have new needs and products have to adapt.

Changes in tech are just one side of the story, the other side is filled with unfortunate actions that can create a bad legacy codebase or a gruesome one. Developers do fixes, slap on new features, one way or the other, usually going beyond what the original architecture had in store. This can happen over the course of several years, and a codebase that was supposed to go in one direction starts sprouting seeds all over the place. It gets bigger and bigger, patched and fixed, until it slowly turns into the monster that another developer has to handle.

There are different layers to a legacy codebase, different spectrums of difficulty. It’s not pleasant, but it can be stub your toe on the door painful, or get hit by a train painful.



Technical debt



When talking about legacy code, technical debt cannot be overlooked. The two might not be equivalent, but they are connected.

How do you get in debt?

It usually starts with something like this:

"We need this done in six months to get to market in time!"

That sentence brings a whole cascade of events that in the end create a monster of a codebase. Market trends influence the way companies build products, the way technology is used and overall, the way codebases are built in time.

It's really hard, near to impossible to be without any technical debt. When companies are started the roadmap only goes to a certain point, there's only so much you can predict and code accordingly. Deadlines for product or feature launches only make things worse, they might be good for the business, but they will have code consequences on the long run.

Growth also happens in stages, if a company aims to become a market leader or wants to create a breakthrough product and prepares accordingly, it will spend a lot more time preparing and might miss its window of opportunity. Instead of becoming a market leader they become a cautionary tale.

For startups, the architecture and design are very much focused on the near future, to get to market as fast as possible, to build a following, to launch a product, to get a foothold in a niche market. If it's an established company, there may be a new functionality or module that needs to be done in the same manner, the same hurried pace.

Overall, technical debt adds up, a sure and fast path to a legacy codebase, the state of which depends a lot on the circumstances - not all legacy code is created equal.

Perl legacy projects

One factor that exists in every discussion about legacy projects - irrelevant of the programming language, is technology evolution. As the environment changes, as the tools, standards and thinking progresses, codebases built in another era become legacy.

In the case of Perl, quite a few of the existent legacy codebases can be traced to the heavy usage of CGI in the 90' and even the early 00'. That's not to say that CGI is to blame, it was the right technology at the time.

Other contributing factors are:

- developers that were not professional programmers
- junior developers without an appropriate ratio of senior developers
- lack of frameworks that required creative solutions which proved untrustworthy on the long run
- lack of maintenance for CPAN modules
- out of date dependencies
- TMTOWTDI

TMTOWTDI is something that deserves a bit more explaining as it's a roadblock in dealing with legacy projects. TMTOWTDI is a principle that allows you more flexibility and freedom in programming, but at the same time, makes life harder for those that come after you. When reading and interpreting someone else's code, the way you would imagine yourself doing it, can be light years away from how someone else actually did it.

In ***Perl Medic: Transforming Legacy Code***, Peter J. Scott said:

"Because there are so many ways to write a Perl program that is not only syntactically correct (Perl makes no objection to running it) but also semantically correct (the program does what it's supposed to—at least in the situations it's been tried in), there is a wide variety of Perl programming styles that you might encounter, ranging from beautiful to what can charitably be described as incomprehensible."

Consistency is a good principle to follow, especially in large scale projects where development goes on constantly, for years or decades. TIMTOWTDIBSCINABTE is a part of that.

Perspective

Working on legacy code is on no ones bucket list, everybody wants a clean or a greenfield project where they can show off their skills and build beautiful code. But life is a bit different.

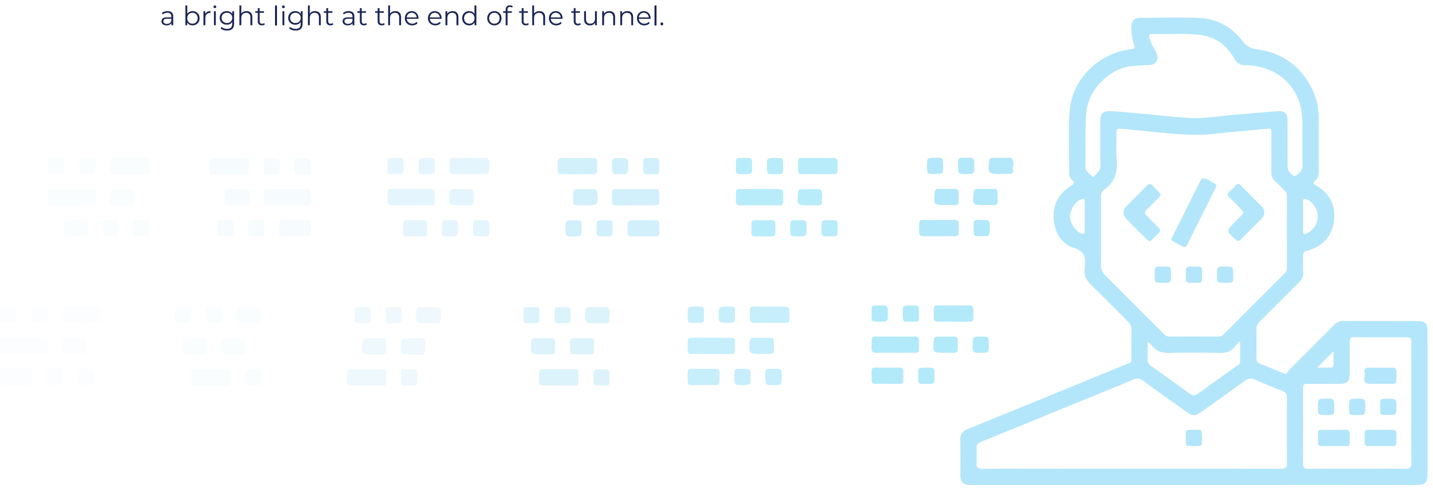
The reaction to legacy code is to utterly hate and curse the developers that "gifted" such a monumental mess, but it might be worth to take a step back and think about it for a minute before reacting strongly to it. It's an obstacle, but it's not a dead end. Yes, it might be three layers of hell deep, but it can be dealt with, one way or another.

There is without a doubt legacy code created by plain bad programming, but there is also legacy code created by people who did not have the knowledge or the resources that we have at hand today. People weren't always Agile, didn't always code to the standard that we have today. Perl, and its environment was not always as good as it is today. At the same time, business decision impacted the way a product was built, superseding developer qualms.

The consideration that in the end all code becomes legacy code, should also dampen heightened spirits.

How was your code a couple of years ago? If you have a look at it, do you feel content, proud or horrified? Is it because you lacked the skill you have now, the time, the resources or the ability to write different code?

The perspective you have when working with legacy code can influence the way you feel about your work and the way you approach it. It also should definitely define the way you create your own future legacy code. There are ways of dealing with legacy projects, they can be fixed, incrementally or totally - there is always a bright light at the end of the tunnel.



Consequences and solutions

All code will one day become legacy. There is no solution in solving this vicious circle, as long as technology keeps moving forward, today's code will become tomorrow's legacy.

We have to deal with the legacy code that exists today and at the same time think about our own future legacy code. There are ways of dealing with the existent legacy code and there are precautions that we may take in order to shape the way code becomes legacy, and make things a bit easier for those that will code after us.

Our legacy Perl working experience proved that improving incrementally is the first response and the best chance to move in the right direction. Refactoring the code is a good option for a legacy codebase, but sometimes a rewrite is needed. Most of the time it's a combination of both.

One thing that we advocate for, is that working code not be thrown away. If it's ugly, it can be prettied up, if it's inefficient, it can be optimized. In large-scale applications throwing away working code should not be an option taken lightly.

Max Kanat-Alexander, in his book ***Code Simplicity*** wrote that you should only rewrite a system if all the following are true:

> You have developed an accurate estimate that shows that rewriting the system will be a more efficient use of time than redesigning the existing system.
>
> You have a tremendous amount of time to spend on creating a new system.
>
> You are somehow a better designer than the original designer of the system or, if you are the original designer, your design skills have improved drastically since you designed the original system.
>
> You fully intend to design this new system in a series of simple steps and have users who can give you feedback for each step along the way.
>
> You have the resources available to both maintain the existing system and design a new system at the same time.

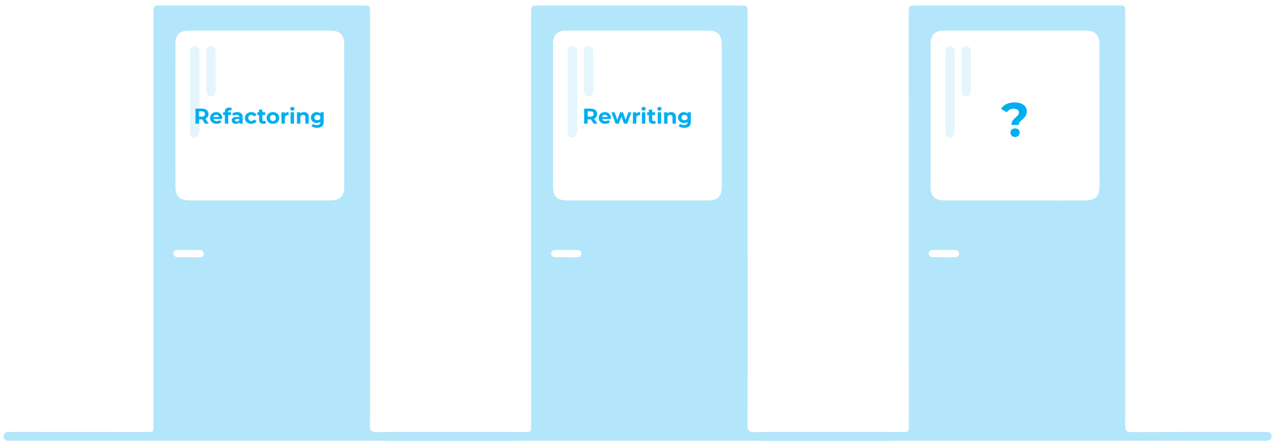
Rewriting and refactoring

These two terms usually come up on different sides of the fence, but it's not really black and white, it's mostly different shades of grey.

Refactoring means making improvements to an existing software without altering its behaviour. If it changes the way the software works from the user's perspective it is no longer refactoring.

Rewriting is, well, rewriting everything, using a different architecture, different business logic.

Going for one or the other is tricky and depends on a lot of things. There is also ***secret door number 3***, the grey area, where you can do both. It really depends on the circumstances.



When you refactor you use the same programming language, making changes that improve the code when it comes to performance and maintenance. When you rewrite you can use the same programming language or a whole new one. For existent Perl projects the usual choice for a rewrite is Python. You can also rewrite a project in the same language. The rewrite can be small, where you redo a few functionalities or complete, where everything is rebuilt from scratch.

The main purpose is to improve the project and comply with the business needs, you use whatever process you can to achieve that. Working code should not be thrown away, if it can be refactored then that's a good option to go on, if not, then a rewrite is the next best thing.

Perl

The process where code becomes legacy is not related to any programming language. However, various factors contribute to transforming a codebase into a legacy codebase and the type of legacy codebase it becomes; including the nature of the programming language, its development, versioning, environment, programming pool and others.

Considering the heightened popularity that Perl had in the 1990's, its footprint remains massive to this day, including under legacy form. A lot of Perl developers today work on legacy systems, our Perl division is no different.

Case study: Perl legacy project rewrite

Dealing with legacy projects, no matter the programming language, is something that every developer goes through at one point. We also had quite a few experiences working with legacy projects on Perl. There is no clear cut way to deal with them, whether it's Perl or another language, you adapt to the circumstances.

Here's one of our experiences working on a Perl legacy project, one that we refactored and rewrote. The decision to rewrite it was taken after client consultation and taking into account the current state of the system and the desired state of the system. Refactoring it completely was not an option as the architecture itself was not suitable for the current needs.

The rewrite project took 9 months for a team of 3 developers.

We rewrote a main part of the business in Perl, an ETL system that had a huge impact on the business processes. Our client's entire product line relied on this system, as such, we had to take into account every single detail that may have an impact on the quality or data integrity.

Making the correlation to legacy code and old code, this was actually a pretty new project, just 4 years old, but with several add ons, fixes and patches over the years. In those 4 years a lot of people worked on the project, so there was a hefty dose of personal marks. None of the folks who actually worked on the code were in the company at the time we worked on it so there was no transfer of knowledge on the coding level, only on the business level.

For the record we changed a lot of the project logic, improving the process itself and also adding a few things. Although it was mostly a rewrite job, we also improved some things that would fall in line with refactoring.

In time the data volume increased substantially and it continued to grow, the system couldn't handle it anymore, the processing time was too long and increasing. The lack of performance was damaging the business.
At the same time, the data complexity also grew, making maintaining it difficult and building on top of it grueling.

The constant patching and developer turnover made it very bug happy, a lot of things were overlooked so the quality of the products that relied on the data processed by the ETL system suffered. The system was really hard to maintain and could not scale to suit the current business needs.

Challenges

The main challenge was rethinking the entire architecture so that it becomes easy to maintain in the future, whilst also remaining scalable. The rewrite was necessary, but we were fully aware that during this process we could also make new mistakes that might hamper other people in the future, so planning ahead and making sure that we did cover every corner was our greatest challenge.

The entire project was data oriented, so understanding the dataset in such a way that we understood any and all implications related to them was crucial. Even though there were no developers around that worked on the existent code, the business analysis people on the client side were very helpful. They also helped us a lot in understanding what we could get rid of, and what should stay in.

When it came to the code itself, things changed a bit. The TMTOWTDI principle and the freedom in synthax made our lives really hard. We needed a lot of time and patience to understand the code and to understand why certain things were done in a certain way. A few of them we gave up on as they were too convoluted. Overall, the code was very hackish.

Another drawback, strictly related to Perl, or the Perl environment, was the fact that several of the CPAN modules that were in use, were also no longer maintained or had little to no documentation. We took a lot of time to analyze the modules, what they did, how they worked to be able to understand their role in the grand scheme of things.

As the system relied on a lot of AWS services, paid services, we also had to balance efficiency and financial cost. Last, but not least, an overall challenge was managing and structuring what at a first glance seemed and was a tremendous amount of work.

Approach

The first step was analyzing in great detail the current implementation, identifying major and minor issues. We managed this by isolating the code logic in pieces, then analyzing it integrated with other system parts. It was a long and time consuming process, but it proved very effective.

Afterwards we regrouped and reimplemented the system logic, by structuring it in many different and separate pieces.

Result

The overall result was a product that was scalable, maintainable, with several improvements in performance. It ticked every box that the client had when we started the project.

Although it was a rough experience for the development team, in the end it had its rewards. Besides the work itself and the result, they also learned a lot of things about clean programming, architecture and the other side of the coin when it comes to flexibility in coding.

Our work was documented in order to make the future legacy code a lot easier to handle for the people that come after us.

Future legacy

As previously mentioned, all code will become legacy. The when and the how depends, that's a separate conversation.

Creating a modern Perl codebase that works well under today's standards, and that will work in the future as well, after developer rotations, standard changes, after tools deprecate and modules are abandoned - is not an easy task. There's only so much you can future proof.

Frameworks won't be maintained, the language may fall out of favour of future, trends will change, times will change and what you write today will be viewed as obsolete.

But developers have control of the code, if you code following good programming principles, if you use unit testing, code reviews, write documentation, your conscience should be clear. Business decisions, technology evolutions and other factors that contribute to a hard to handle legacy codebase have their share to blame, but that's something that's usually out of reach of the developer.

There will always be legacy code, the important thing is to tackle it better, to make sure that you avoid creating another mind numbing legacy code in the future. It will still be legacy code one day, but it's a different story if it's just bad or darn right impossible.

Evozon is a software development company founded in 2005 as a Perl and Java shop. We currently cover a wide-range of technologies and industries and have more than 500 personnel. Our Perl division is one of the largest service based teams in Europe and through an expansion to Python, we can now cover Perl, Python and mixed technology projects, including legacy code transitions.

Samuel Andras
Business Development

samuel.andras@evozon.com
evozon.com